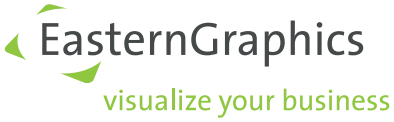

Useful OFML methods for OAP data

Version 1.11

Thomas Gerth, EasternGraphics GmbH

November 6, 2023

Contents

References

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

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

[oap] OAP – OFML Aided Planning. Version 1.5. 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

The specifications are available via the pCon Download Center in the category *OFML Specications*:
https://download-center.pcon-solutions.com

1 Introduction

Calls to OFML methods are used in OAP data creation for three purposes:

1. Realization of product logics as an action, which is bound to an interactor
2. Obtaining information about objects within expressions using syntax construct *methodCall()*, e.g. for
   - validity conditions of interactors and actions
   - determining the position of interactor symbols
   - determining arguments for a *MethodCall* action
3. Specifying (target) objects via object category *MethodCall*

While in some cases specially programmed methods from project-specific classes have to be used, in other cases methods can be used which are implemented in OFML base classes (i.e., requiring no special programming).

This document lists and describes some useful methods of base classes that would otherwise have to be gathered from multiple specifications or documents. In addition to the specification of each method, there also is given an example of an application in OAP table *MethodCall*.

Notes:

- Interfaces *Base*, *Complex*, *Property* [property] and *Article* [article] all are implemented in base class *OiPlElement*. As this is the base class for all classes used to represent articles, all methods described for these interfaces can be used with/for all article instances (see also appendix B).
- This document describes only the methods of the mentioned interfaces and classes that are usefull for *MethodCall* actions in OAP. For a full documentaion see the according specifications.

This document version reflects the state of OFML base packages OI version 1.43.0, XOI version 1.60.0 and GO version 1.17.34 contained in the application releases of Fall 2023.

2 Interface *Base* (incl. *MObject*)

2.1 Class and Category

- *getClass()* → *String*

  The function returns the name of the direct type of the implicit instance.

  ```
  MC_GET_CLASS;Instance;@IF_Base;getClass;
  ```

- *isCat(pCat(Symbol))* → *Int*

  The function returns 1 if the implicit instance belongs to the specified category.

  ```
  MC_IS_CAT_FOOBAR;Instance;@IF_Base;isCat;@FooBar
  ```

- *hasMember(pName(Symbol))* → *Int*

  The function returns 1 if the class of the implicit instance has the specified member (variable or method).

  This is useful if neither *getClass()* nor *isCat()* can be used to determine whether a certain method can be called on the target object.

  ```
  MC_HAS_GET_MTID;Instance;@IF_Base;hasMember;@getMTID
  ```

  (For method *getMTID()* see section 6.)

2.2 Object hierarchy

- *getFather()* → *MObject*

  The function returns a reference to the father object. If the implicit instance does not have a father, the result is `NULL`[1].

  ```
  MC_GET_FATHER;Instance;@IF_Base;getFather;
  ```

- *getRoot()* → *MObject*

  The function returns a reference to the root instance of the hierarchy in which the implicit instance is located.

  ```
  MC_GET_ROOT;Instance;@IF_Base;getRoot;
  ```

Note:
As object references of OFML instances cannot be used in OAP, these methods can be employed only for comparison with `NULL` (in the case of *getFather()*) or for comparing the results of both methods or in combination with object category *MethodCall*.

E.g., a recurring requirement in OAP data creation is, that an interactor should not be displayed, if the article instance is on the top planning level (i.e. if it is not an element of a planning group). With the 2 methods above the corresponding condition is:

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

Or, if the condition should be linked with a specific condition in the case, that the article instance is an element of a planning group (see also section 8):

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

2.3 Spatial model

- *getTrAxis()* → *Int*

  The function returns the current degrees of freedom of translation of the implicit instance with respect to the 3 individual axes of local coordinate system.

  The return value results from the addition of allowed axes, where x, y and z-axis are represented by 1, 2 resp. 4. (I.e., if return value is 0, the object cannot be moved.)

  ```
  MC_GET_TRANSLATION_AXIS;Instance;@IF_Base;getTrAxis;
  ```

- *getRtAxis()* → *Int*

  The function returns the current degrees of freedom of rotation of the implicit instance with respect to the 3 individual axes of local coordinate system.

  The return value results from the addition of allowed axes, where x, y and z-axis are represented by 1, 2 resp. 4. (I.e., if return value is 0, the object cannot be rotated.)

  ```
  MC_GET_ROTATION_AXIS;Instance;@IF_Base;getRtAxis;
  ```

[1] This may be true only if the implicite instance is the root instance of an object hierarchy.

3 Interface *Complex*

- *getWidth()* → *Float*

  The function returns the (nominal) width of the implicit instance.

  ```
  MC_GET_WIDTH;Instance;@IF_Complex;getWidth;
  ```

- *getHeight()* → *Float*

  The function returns the (nominal) height of the implicit instance.

  ```
  MC_GET_HEIGHT;Instance;@IF_Complex;getHeight;
  ```

- *getDepth()* → *Float*

  The function returns the (nominal) depth of the implicit instance.

  ```
  MC_GET_DEPTH;Instance;@IF_Complex;getDepth;
  ```

Typically, the objects possess some properties related to dimensions. Instead of calling the methods above, these properties can – and should be – used!

This avoids possible pitfalls due to the actual implementation of these methods in the class of the implicit instance. Consider the following notes on this regard:

- It depends on the individual class of the implicit instance whether these methods return the exact dimension according to the geometry of the object, or some other values (e.g. nominal values according to the price list).
- The default implementations in the base class *OiPlElement* behave as follows:
  If a value was assigned for the width/height/depth via the according *set-method*, this value will be returned. If no value was assigned via the set-method, the dimension is determinded based on method *getLocalBounds()* of interface *Base* [ofml] and this dimension will be returned until (a possible) succeeding call of the set-method[2].
  This means, if no value was assigned until the first call of the *get-method* the according dimension will be determined at that time based on *getLocalBounds()* and afterwards all calls of the get-method will return this value (unless the dimension will be explicitely changed via the set-method)!
  So, these implementations *cannot* be used if, e.g., the dimension may be changed by the user via an according property and the new dimension is not be assigned via the set-method!
- The implementations in the base class *GoMetaType* delegate to the main child (if any), otherwise use the inherited implementation as described above (class *OiPlElement*).
  So, these implementations can be used, if the main child defines the according dimension of the metatype instance and the dimension may be changed by the user only via a metatype property causing a change of the main child due to a changed basic article number[3].
- The implementions in the base class for all planning groups *xOiPlGroup* determine the according dimension based on method *getLocalBounds()*, i.e., they cannot be changed via the set-method, but reflect the dimensions of the current group elements.
  Thus, in the most cases these implementations can be safely used to determine the dimensions of the group. This is important, because the group instances typically do not have dimension properties which could be used otherwise.

[2] Actually, according to the specification in [ofml], the methods should return a value determinded based on *getLocalBounds()* only, if no value was assigned via the set-method.

[3] According to the described behaviour of the default implementations in *OiPlElement*, the first call of the get-method after the article change then returns the right dimension value.

4 Interface *Property*

- *hasProperties() → Int*

  The function returns 1, if properties are defined for the implicit instance, otherwise it returns 0.

  ```
  MC_HAS_PROPERTIES;Instance;@IF_Property;hasProperties;
  ```

- *hasProperty(pKey(Symbol)) → Int*

  The function returns 1 if a property with indicated key is defined for the implicit instance, otherwise 0.

  ```
  MC_HAS_PROP_FOOBAR;Instance;@IF_Property;hasProperty;@FooBar
  ```

- *getPropState(pKey(Symbol)) → Int*

  The function returns the current activation state of the property of the implicit instance with the specified key according to the old interface.

  The possible state values are:

  -1 non-active and non-visible

  0 non-active but visible

  1 active

  If no property with the specified key is defined for the implicit instance, the return value is -1.

  ```
  MC_GET_PROPSTATE_FOOBAR;Instance;@IF_Property;getPropState;@FooBar
  ```

- *getPropState2(pKey(Symbol)) → Int*

  The function returns the current activation state of the property of the implicit instance with the specified key according to the new interface.

  The possible state values are:

  0 not visible (and not editable)

  1 visible, but not editable

  3 visible and editable

  If no property with the specified key is defined for the implicit instance, the return value is 0.

  ```
  MC_GET_PROPSTATE2_FOOBAR;Instance;@IF_Property;getPropState2;@FooBar
  ```

- *checkPropValue(pKey(Symbol), pValue(Any), ...) → Int*

  The method checks whether the given value can be assigned to the property of the implicit instance with the specified key.

  The return value is `1` if the implicit instance has the specified property and if all checks have been successfully completed, otherwise `0`.

  If an additional optional parameter is passed, it specifies whether the passed value, if proven valid, should be compared to the current value of the property (`1`) or not (`0`, default).
  If comparison is enabled, the method returns `0` if the passed value is equal to the current value of the property.

  See the interface specification [property] for a list of checks performed by the standard implementation.

  ```
  MC_CHECK_PROPVALUE_FOOBAR_VAL1;Instance;@IF_Property;checkPropValue;@FooBar,@Val1
  ```

5 Interface *Article*

- *getArticleSpec()* → *String* | *Void*

  The function returns the base article number of the implicite instance, or `NULL`, if the instance does not represent an article.

  Note:
  The function is not applicable if the actual article instance is encapsulated by a metatype instance!

  ```
  MC_GET_ARTICLE;Instance;@IF_Article;getArticleSpec;
  ```

6 Class *GoMetaType*

- *getMTID()* → *String*

  Returns the ID of the metatype represented by the implicite instance.

  ```
  MC_GET_MT_ID;Instance;::ofml::go::GoMetaType;getMTID;
  ```

  If it is unproved, that the class of the target object is *::ofml::go::GoMetaType* (or a derived class) method *hasMember()* (see section 2.1) should be used to check this, as shown in the following condition sample:

  ```
  methodCall("AC_call_HAS_GET_MT_ID") && methodCall("AC_call_GET_MT_ID") == "FOO_BAR"
  ```

- *getChID()* → *Symbol* | *Void*

  Returns the child ID under which the implicite Meta type instance is registered in a father Meta type instance (if any).

  If the implicite Meta type instance is not a child of another Metatype instance, return value is `NULL`.

  This refers to colum 1 (*child_key*) in table *go_children*.

  ```
  MC_GET_CHILD_ID;Instance;::ofml::go::GoMetaType;getChID;
  ```

  This can be used, e.g., to display an interactor on a Meta type instance only if it was created as a child of a parent Meta type instance, as shown in the following interactor condition sample:

  ```
  methodCall("AC_call_GET_CHILD_ID") != NULL
  ```

  Or the interactor should only be displayed if the Meta type instance was created at a certain position in the geometry of the parent Meta type instance:

  ```
  methodCall("AC_call_GET_CHILD_ID") == @FB_Left_Sreen
  ```

7 Planning group classes

The XOI base library provides 4 types of planning groups, implemented in according classes:

- *xOiJointPlGroup* realizes a joint planning, i.e. new elements will be added either to the right or to the left of the already existing group elements.
- *xOiLayouGroup* realizes a group with a free but continuous (2-dimensional) layout consisting of branches.
- *xOiTabularPlGroup* realizes a group with a tabular layout.
- *xOiCustomPlGroup* supports the realization of groups with specific (custom) planning logics.

The subsequent sections describe the concept and the specific methods of these classes. However, all planning group classes also provide some common functions, which are covered here in this section in according subsections:

- Elements of a planning group may be assigned to self-defined categories (see 7.2).
- Controlled by options in the control data table [an0601], properties may be pulled from the element level to the planning group level. The according properties on planning group level are called *common properties*. Subsection 7.3 covers methods related to common properties.

Note: The examples in these subsections use class *xOiLayoutGroup*. If necessary, this has to be replaced by the class actually used.

7.1 General Notes

- If the interactor is bound to the planning group itself, the target object for the action, performing a call of one of the methods described in the following sections, has to belong to category *Self*. Otherwise, if the interactor is bound to a group element, the target object has to belong to category *ParentArticle* (see section "Object definitions" in [oap]).
- Some methods require a group element as a parameter.

  If the interactor is bound to a group element, placeholder *$SELF* can be used for such a parameter. Alternativly, or if the interactor is bound to the planning group, the element has to be specified via expression *methodCall()* using methods accessing specific group elements (and not requiring a group element as a parameter).

  The examples in the following sections use placeholder *$SELF*, i.e., it is assumed, that the corresponding interactor is bound to a group element.
- Some methods return object references.

  As references to OFML instances cannot be used in OAP, these methods can be employed only for comparison with `NULL` or in combination with object category *MethodCall*!

  > E.g., assuming a *MethodCall* action with identifier `AC_call_NEIGHBOR_R` calling a method which returns the neighbor to the right (or `NULL` if there is no neighbor at that side), this action can be used to write the condition
  >
  > ```
  > methodCall("AC_call_NEIGHBOR_R") == NULL
  > ```
  >
  > This condition is satisfied, if the active object has no neighbor to the right side.

7.2 Element categories

- *assignElementCat(pObj(MObject), pCat(Symbol))* → *Void*

  Assigns given group element to specified category.

  Instead of an group element, also an instance can be passed that represents an immediate subarticle of one of the group elements. (Below, these instances are also called *grand subarticles*.)

  If a selectability state was/is specified for the category (see method *setSelectability4ElemCat()* below), this state will be applied to the given object.

  Note:
  If the object is up to be removed, the internal data structure containing the information regarding the assigned element categories will be updated automatically, i.e. it is not necessary to call *removeElementCat()* (see below) before the removal.

  ```
  MC_ASSIGN_CAT_SEAT;Instance;::ofml::xoi::xOiLayoutGroup;assignElementCat;$SELF,@Seat
  ```

- *removeElementCat(pObj(MObject), pCat(Symbol | Void)) → Void*

  Removes given group element from specified category.

  Instead of an group element, also an instance can be passed that represents an immediate sub-article of one of the group elements.

  If parameter *pCat* is NULL, the element will be removed from all assigned categories.

  Note:
  If a selectability state was specified for the/a category (see method *setSelectability4ElemCat()* below) and applied accordingly to the given object, its state will not be changed at this moment!
  (If necessary, this has to be done by the client.)

  ```
  MC_REMOVE_CAT_SEAT;Instance;::ofml::xoi::xOiLayoutGroup;removeElementCat;$SELF,@Seat
  ```

- *clearElementCat(pCat(Symbol)) → Void*

  Clears specified category.

  ```
  MC_CLEAR_CAT_SEAT;Instance;::ofml::xoi::xOiLayoutGroup;clearElementCat;@Seat
  ```

- *isElementCat(pObj(MObject), pCat(Symbol)) → Int*

  Returns True (1) if given instance is a group element or a grand sub-article instance and if it belongs to specified element category.

  ```
  MC_IS_CAT_SEAT;Instance;::ofml::xoi::xOiLayoutGroup;isElementCat;$SELF,@Seat
  ```

- *setSelectability4ElemCat(pCat(Symbol), pState(Int)) → Void*

  Sets the selectability state for all group elements or grand sub-article instances belonging to specified category.

  Parameter *pState* may have the following values:

  0 the objects cannot be selected

  1 the objects can be selected

  If an additional optional parameter is given, it specifies whether the state should be applied hierarchically to child objects of the affected category objects (1) or not (0, default)[4].
  (For a given category the same optional parameter should be used for both states.)

  ```
  MC_CAT_SEAT_SELECTABLE;Instance;::ofml::xoi::xOiLayoutGroup;setSelectability4ElemCat;\
  @Seat,1
  ```

7.3 Methods related to common properties

- *assignCommonPropValues(pEl(MObject)) → Void*

  Assigns current values of common group properties to given layout element (insofar it possesses these properties).

  ```
  MC_ASSIGN_COMMON_PROPS;Instance;::ofml::xoi::xOiLayoutGroup;assignCommonPropValues;\
  $SELF
  ```

- *updateCommonProperties() → Void*

  The function updates the common properties.

  To be called by clients if a property has been changed which affects the visibility or the state of common properties, where the changed property itself is not included in the set of common properties.

  ```
  MC_UPDATE_COMMON_PROPS;Instance;::ofml::xoi::xOiLayoutGroup;updateCommonProperties;\
  $SELF
  ```

[4] In case of value 1, methods *hierSelectable()* resp. *notHierSelectable()* (OFML interface *Base*) will be used (instead of *selectable()* resp. *notSelectable()*) to apply the passed state.

7.4 Other common functions

- *isGrandSubArticle(pObj(MObject), ...) → Any*

  Checks whether given instance is a sub-article of one of the group elements[5].

  Additional optional parameter can be used to control the type of return value:

  – If it is 0 (default), the return value is 0 (false) or 1 (true).

  – If it is 1, return value is NULL, if given instance is not a grand sub-article, otherwise a *Vector* containing the following information:

  1. the group element being the superior article
  2. the sub-article ID of the grand sub-article

  Note:
  In the context of OAP method calls only the default behavior is useful.

  ```
  MC_IS_GRAND_SUBARTICLE;Instance;::ofml::xoi::xOiLayoutGroup;isGrandSubArticle;$SELF
  ```

  This example assumes object category *TopArticle* for the target object.

- *checkElDistance(pOp(Symbol), pCheckVal(Float), pObj(MObject), pAttPt(Symbol |Any), ...) → Int*

  Compares the distance from given attach point of specified group element to other objects in the scene with a specified value.

  Parameter *pAttPt* either is the key of an attach point of the given instance or a specific attach point definition related to the given instance.
  If an attach point definition is given, the 4th element (mode) can be omitted. Mode *@Sibling* is then assumed.

  If an additional optional parameter is given, it specifies a Vector of 3 *Float* values, defining the offset to be applied to the coordinates (x, y, z) of the attach point position.
  This may be useful/necessary in order to modify the position of a given attach point to get the desired result[6].

  The distance is determined based on global OFML function *oiGetDistance()* ([ofml], section 6.10): The starting point of the search ray is the (possibly modified) position of the attach point translated into world coordinates. The direction of the search ray is determined based on the global PY-rotation of the given instance and the PY-rotation to be applied to a sibling element according to the attach point definition.
  If no intersection of the search ray with an object in the scene was found, the distance is undefined. Otherwise, it is defined as the distance along the search ray from the starting point to the first point of intersection.

  Note: The (possibly modified) position of the attach point should not be inside of the geometry of the given instance. Otherwise, the search ray may intersect with one of the instance's own geometry objects!

  The following comparison operations (parameter *pOp*) may be applied[7]:

  `@GE` The comparison is successful if no intersection was found or if the distance is greater than or equal to the check value.

  `@GT` The comparison is successful if no intersection was found or if the distance is greater than the check value.

  `@LE` The comparison is successful if an intersection was found and if the distance is lower than or equal to the check value.

  `@LT` The comparison is successful if an intersection was found and if the distance is lower than the check value.

[5] Only those group elements will be checked for sub-articles which represent articles themselves.
[6] A typical usage scenario is to adapt the Y-coordinate, as the the standard value 0.0 may not result in an intersection of the search ray with a scene object.
[7] The comparisons apply an epsilon of 0.0001.

In case of errors related to the parameters or if the comparison is not successful, the return value is False (0), otherwise True (1).

```
MC_CHECK_DISTANCE_ADD_B2;Instance;::ofml::xoi::xOiLayoutGroup;checkElDistance;\
@GT,1.995,$SELF,@AP_Add_B2,[0.0, 0.4, 0.0]
```

8 Class *xOiJointPlGroup*

Instances of *xOiJointPlGroup* realize a joint planning, i.e., new elements/articles will be added either to the right or to the left of the already existing elements[8].

If the *insertion mode* (see option *@InsertMode* in [an0601]) has the (default) value 0, elements may not be inserted between two existing group elements resp. may not be removed from the middle of the group. In this case, interactors for adding resp. removing elements have to be equipped with according validity conditions.
For that purpose, methods of class *xOiJointPlGroup* can be used which concern the so-called *topological order list* of the group, i.e. the order of the group elements from the left-most to the right-most element (see 8.1).

Some aspects of the behaviour may be controlled by specifying according options in control data table `jointplgroup` [an0601].

8.1 Accessing topological order list

- *getElCount()* → *Int*

  Returns the number of elements in the topological order list.

  ```
  MC_ELEM_COUNT;Instance;::ofml::xoi::xOiJointPlGroup;getElCount;
  ```

- *isEmpty()* → *Int*

  Returns True (1), if the topological order list is empty.

  ```
  MC_IS_EMPTY;Instance;::ofml::xoi::xOiJointPlGroup;isEmpty;
  ```

- *isNotEmpty()* → *Int*

  Returns True (1), if the topological order list is not empty.

  ```
  MC_IS_NOT_EMPTY;Instance;::ofml::xoi::xOiJointPlGroup;isNotEmpty;
  ```

- *getElPos(pObj(MObject))* → *Int* | *Void*

  Returns the position of given element in the topological order list.

  Return value is an integer specifying the position starting with 0 (for the left-most element) or `NULL` if the object ist not contained in the topological order list.

  ```
  MC_GET_EL_POS;Instance;::ofml::xoi::xOiJointPlGroup;getElPos;$SELF
  ```

- *firstObj()* → *MObject* | *Void*

  Returns the left-most group element (or `NULL` if group is empty).

  (Useful in combination with object category *MethodCall*.)

  ```
  MC_FIRST_OBJ;Instance;::ofml::xoi::xOiJointPlGroup;firstObj;
  ```

[8] In fact, *xOiJointPlGroup* itself is derived from base class *xOiLRPlGroup* and many of the methods described below are inherited from that class.

- *lastObj() → MObject | Void*

  Returns the right-most group element (or `NULL` if group is empty).

  (Useful in combination with object category *MethodCall*.)

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

- *isFirstObj(pObj(MObject)) → Int*

  Returns True (1), if given element is the left-most group element.

  ```
  MC_IS_FIRST_OBJ;Instance;::ofml::xoi::xOiJointPlGroup;isFirstObj;$SELF
  ```

- *isLastObj(pObj(MObject)) → Int*

  Returns True (1), if given element is the right-most group element.

  ```
  MC_IS_LAST_OBJ;Instance;::ofml::xoi::xOiJointPlGroup;isLastObj;$SELF
  ```

- *neighbor(pObj(MObject), pDir(Symbol)) → MObject | Void*

  Returns the neighbor of given group element in given direction (`@L` – left, `@R` – right).

  Returns `NULL` if group is empty or if there is no neighbor in given direction.

  ```
  MC_NEIGHBOR_R;Instance;::ofml::xoi::xOiJointPlGroup;neighbor;$SELF,@R
  ```

8.2 Other methods

- *replaceElement(pEl(MObject), pPID(Symbol), pArticle(String), pOptions(Any)) → MObject | Void*

  Replaces given group element by an instance of specified article[9].

  If parameter *pEl* is `NULL` and the group contains only a single topological element, this element will be replaced.

  Parameter *pPID* specifies the OFML program ID.
  Parameter *pArticle* specifies the base article number.

  Parameter *pOptions* either is `NULL` or a Vector of additional options for article creation:

  1. Varcode type (Symbol: @VarCode, @OFMLVarCode, @Final)
  2. Varcode (String)
  3. PropValues (Vector of [key, value] pairs)
  4. PropStates (Vector of [key, state] pairs)

  All elements in this Vector may be `NULL`. Elements 3 and 4 may be omitted.

  The new element gets the same position and rotation as the replaced element.

  Returns `NULL` if the new element could not be created, otherwise the reference to the new article instance. (The new instance gets the same object name as the replaced instance.)

  ```
  MC_CREATE_XY;Instance;::ofml::xoi::xOiJointPlGroup;replaceElement;\
  $SELF,@foo_bar,"XY",NULL
  ```

  Notes:

  – If the new element has another width than the replaced one, neighbors of the element should be moved according to the changed width and/or the element itself should be re-positioned. This re-positioning procedure has to be enabled by specifying an according value for option *@InsertMode* in the control data table [an0601].

[9] This method can be used in situations where there is no suitable property for an action of type *PropEdit* (or *PropChange*) in order to change the article of the active object.

– If the interactor with the *MethodCall* action is bound to a group element (not to the planning group instance), after the 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!

- *replaceElement2(pEl(MObject), pPID(Symbol), pArticle(String), pOptions(Any), pAttPt(Symbol), ...) → MObject | Void*

  Replaces given group element by an instance of specified article.

  The behavior is almost identical with *replaceElement()* (see above) except for the positioning of the new element:
  It will be positioned using a given attach point of left or right neighbor. The key of the attach point is specified in parameter *pAttPt*. Per default, the left neighbor is used as the reference object. However, the reference object may be explicitely specified in additional optional parameter passing the according direction (Symbol: `@L` or `@R`).

  ```
  MC_CREATE_XY_2;Instance;::ofml::xoi::xOiJointPlGroup;replaceElement2;\
  $SELF,@foo_bar,"XY",NULL,@AP_LF
  ```

- *isBusyAttPt(pEl(MObject), pAttPt(Symbol)) → Int*

  Returns True (1) if given group element possesses an OFML attach point with specified key and if there is a sibling or child element at this attach point.

  ```
  MC_IS_ATTPT_R_BUSY;Instance;::ofml::xoi::xOiJointPlGroup;isBusyAttPt;$SELF,@ATTPT_R
  ```

9 Class *xOiLayoutGroup*

Instances of *xOiLayoutGroup* realize a planning group with a free but continuous (2-dimensional) layout consisting of branches.

In contrast to *xOiJointPlGroup* (see previous section) this class supports arbitrary types of connections, e.g. "T" and "X" connections. For that purpose, the notion of the *topological order list* in *xOiJointPlGroup* here is replaced by the notion of the *layout*.

There are two types of group elements:

- Elements defining the *layout* of the group.

  These elements have neighbor-relationships. They are attached to other layout elements by means of OFML *attach points*. Two layout elements are called *neighbors* if they were connected (placed next to each other) using a pair of attach points[10].

  Methods regarding layout elements are described in 9.1.

- *Additional* (non-layout) elements.

  Typically, they are created based on the current layout (defined by the layout elements), e.g. frames for sofa arrangements, links between layout elements and alike.

  Methods regarding additional elements are described in 9.5.

There are special layout elements called *fork elements*: They either are elements where the layout forks (i.e. elements with more then two neighbors, such as "T" or "X" connections) or are elements otherwise playing a special role in the layout. Fork elements are determined automatically when a layout element is added or removed[11]. Alternatively, layout elements may be marked or unmarked as fork elements using methods *markAsForkElement()* resp. *unMarkAsForkElement()* (see9.2).
Methods regarding fork elements are described in 9.2.

[10] Note: two layout elements may be located next to each other but be no neighbors because they were not placed next to each other using a pair of attach points.

[11] As long as a layout element has more than 2 neighbors it is defined to be a fork element.

By default, layout elements may be added only to layout elements with a free attach point. Accordingly they may be removed only if they have only one attached neighbor. However, if the *insertion mode* is 2 (see option *@InsertMode* in the control data table [an0601]) insertion and removal is allowed anywhere (implying a possible repositioning of neighbors).

By default, it is not allowed/supported to change the dimensions of individual layout elements. However, if the insertion mode is greater than 0, it is allowed to change the dimension of a layout element (implying a possible repositioning of neighbors).

The layout consists of one or more *branches*: It starts with the so-called *root branch*. It grows when a *successor* element is attached to the last branch element or a *predecessor* element is attached to the first branch element.

When a new layout element is attached to an element of the root branch, which already has 2 neighbors (fork element), a new *child branch* is created. A new branch also is started, if the new element is attached to a first or last element of the branch and if that element is marked as a fork element. Alternatively, by calling method *startNewBranch()* (see section 9.4) right before adding a new layout element, a new branch explicitly can be started, even if the predecessor does not have 2 neighbors (or more).

Elements of a branch where a child branch sets off are called *nodes*. A node element always also is a fork element, but, vice versa, a fork element does not necessarily have to be a node element.

The root branch can grow in both directions. Whether other branches also may grow in both directions or only in one direction is specified by option *@ChildBranchGrowthMode* in control data table `layoutgroup` [an0601].

Branches are identified by numbers in the order of creation starting with 1[12].
Methods regarding branches are described in 9.3.

Some aspects of the behaviour may be controlled by specifying according options in control data table `layoutgroup` [an0601].

9.1 Layout elements

Note:
The order of the elements in the list of layout elements reflects the order of insertion resp. removal of the elements!

- *isLayoutEmpty()* → *Int*

  Returns True (1) if the group does not contain any layout element.

  ```
  MC_IS_LAYOUT_EMPTY;Instance;::ofml::xoi::xOiLayoutGroup;isLayoutEmpty;
  ```

- *isLayoutElement(pObj(MObject))* → *Int*

  Returns True (1) if given instance is a layout element.

  ```
  MC_IS_LAYOUT_EL;Instance;::ofml::xoi::xOiLayoutGroup;isLayoutElement;$SELF
  ```

- *firstLayoutElement()* → *MObject* | *Void*

  Returns the first entry from the list of current layout elements (or `NULL` if the list currently is empty).

  ```
  MC_FIRST_LAYOUT_EL;Instance;::ofml::xoi::xOiLayoutGroup;firstLayoutElement;
  ```

[12] As child branches can disappear during the process of layout creation, the order of the branch numbers can be discontinuous.

- *lastLayoutElement() → MObject | Void*

  Returns the last created element layout element (or `NULL` if the list of layout elements currently is empty).

  ```
  MC_LAST_LAYOUT_EL;Instance;::ofml::xoi::xOiLayoutGroup;lastLayoutElement;
  ```

- *getNeighbor(pObj(MObject), pAttPt(Symbol)) → MObject | Void*

  Returns the neighbor of given layout element at given attach point.

  Returns `NULL` if given instance is not a layout element or if there is no neighbor at given attach point.

  See also method *getAdjacentElement()* below.

  ```
  MC_NEIGHBOR_R;Instance;::ofml::xoi::xOiLayoutGroup;getNeighbor;$SELF,@ATTPT_R
  ```

- *hasNeighbors(pObj(MObject)) → Int*

  Returns True (1) if given instance is a layout element and if it currently has neighbors.

  ```
  MC_HAS_NEIGHBORS;Instance;::ofml::xoi::xOiLayoutGroup;hasNeighbors;$SELF
  ```

- *getAdjacentElement(pObj(MObject), pAttPt(Symbol)) → MObject | Void*

  Returns the group element which occupies the specified attach point of given layout element.

  In contrast to *getNeighbor()* (see above) this method also considers elements *not* being a neighbor of given layout element, i.e. which are not explicitely connected to it but are just located beside it occupying the specified attach point geometrically.

  Returns `NULL` if given instance is not a layout element or if there is no attach point currently defined with given key or if there is no group element occupying the specified attach point.

  Note:
  The returned group element does not necessarily have to be a layout element.

  ```
  MC_ADJACENT_R;Instance;::ofml::xoi::xOiLayoutGroup;getAdjacentElement;$SELF,@ATTPT_R
  ```

- *canBeRemovedFromLayout(pObj(MObject)) → Int*

  Returns True (1) if given instance is a layout element and if it can be deleted (removed from the layout structure).

  A layout element can be removed (by the definition of this implementation) if the insert mode is 2 (see option *@InsertMode*) or if it has only one neighbor.

  ```
  MC_CAN_REMOVE;Instance;::ofml::xoi::xOiLayoutGroup;canBeRemovedFromLayout;$SELF
  ```

- *hasFreeAttPts2(pObj(MObject), pCheckAll(Int)) → Int*

  Returns True (1) if given instance is a layout element and if it has at least one free OFML attach point, otherwise False (0).

  An attach point is free, if it was not used to connect with a neighbor layout element and if there is no (other) sibling or child element occupying this attach point geometrically.

  Parameter *pCheckAll* specifies whether all attach points of given instance have to be checked (1) or only the additional attach points (0, i.e. ignoring the standard attach points).

  ```
  MC_HAS_FREE_ATTPTS;Instance;::ofml::xoi::xOiLayoutGroup;hasFreeAttPts2;$SELF,0
  ```

- *isFreeAttPt(pObj(MObject), pAttPt(Symbol)) → Int*

  Returns True (1) if given instance is a layout element and if specified OFML attach point (key) currently is defined for the given instance and if it is free.
  Otherwise returns False (0).

  ```
  MC_IS_ATTPT_R_FREE;Instance;::ofml::xoi::xOiLayoutGroup;isFreeAttPt;$SELF,@ATTPT_R
  ```

9.2 Fork elements

- *hasForkElements() → Int*

  Returns True (1) if the group contains fork elements.

  ```
  MC_HAS_FORKS;Instance;::ofml::xoi::xOiLayoutGroup;hasForkElements;
  ```

- *isForkElement(pObj(MObject)) → Int*

  Returns True (1) if given instance is a fork element.

  ```
  MC_IS_FORK;Instance;::ofml::xoi::xOiLayoutGroup;isForkElement;$SELF
  ```

- *markAsForkElement(pObj(MObject)) → Int*

  Marks given layout element as a fork element.

  Has no effect, if given instance is not a layout element of the group or if it already is determined/marked as a fork element. In this case, return value is 0, otherwise 1.

  ```
  MC_MARK_AS_FORK;Instance;::ofml::xoi::xOiLayoutGroup;markAsForkElement;$SELF
  ```

- *unMarkAsForkElement(pObj(MObject)) → Int*

  Unmarks given layout element as a fork element.

  Has no effect, if given instance is not a fork element or if it is a node element. In this case, return value is 0, otherwise 1.

  ```
  MC_UNMARK_AS_FORK;Instance;::ofml::xoi::xOiLayoutGroup;unMarkAsForkElement;$SELF
  ```

9.3 Branches

- *getLatestBranch() → Int*

  Returns the number of the branch created most recently.

  ```
  MC_GET_LAST_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;getLatestBranch;
  ```

- *isLatestBranch(pBranch(Int)) → Int*

  Returns True (1) if there is a branch with given number and if this branch was created most recently.

  ```
  MC_IS_ROOT_BRANCH_LAST;Instance;::ofml::xoi::xOiLayoutGroup;isLatestBranch;1
  ```

- *getBranch(pObj(MObject)) → Int*

  Returns the number of the branch the given layout element belongs to.

  If given object is not a layout element return value is 0.

  ```
  MC_GET_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;getBranch;$SELF
  ```

- *isBranchElement(pBranch(Int), pObj(MObject)) → Int*

  Returns True (1) if given layout element belongs to the specified branch.

  ```
  MC_IS_ROOT_BRANCH_EL;Instance;::ofml::xoi::xOiLayoutGroup;isBranchElement;1,$SELF
  ```

- *isNodeElement(pObj(MObject)) → Int*

  Returns True (1) if given object is a layout element and if it is a node element of the branch it belongs to.

  ```
  MC_IS_ROOT_BRANCH_NODE;Instance;::ofml::xoi::xOiLayoutGroup;isNodeElement;1,$SELF
  ```

- *getFirstBranchElem(pBranch(Int)) → MObject | Void*

  Returns the first element from the topological order list of given branch.

  Returns `NULL` if there is no branch with given number.

  ```
  MC_FIRST_EL_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;getFirstBranchElem;1
  ```

- *isFirstBranchElem(pBranch(Int), pObj(MObject)) → Int*

  Returns True (1) if given layout element belongs to the specified branch and if it is the first element in the topological order list of that branch.

  ```
  MC_IS_FIRST_EL_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;isFirstBranchElem;\
  1,$SELF
  ```

- *getLastBranchElem(pBranch(Int)) → MObject | Void*

  Returns the last element from the topological order list of given branch.

  Returns `NULL` if there is no branch with given number.

  ```
  MC_LAST_EL_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;getLastBranchElem;1
  ```

- *isLastBranchElem(pBranch(Int), pObj(MObject)) → Int*

  Returns True (1) if given layout element belongs to the specified branch and if it is the last element in the topological order list of that branch.

  ```
  MC_IS_LAST_EL_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;isLastBranchElem;\
  1,$SELF
  ```

- *isFirstOrLastBranchElem(pBranch(Int), pObj(MObject)) → Int*

  Returns True (1) if given layout element belongs to the specified branch and if it is the first or the last element in the topological order list of that branch.

  ```
  MC_IS_END_EL_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;isFirstOrLastBranchElem;\
  1,$SELF
  ```

- *getBranchPredecessor(pObj(MObject)) → MObject | Void*

  Returns the topological predecessor branch element of given layout element.

  Returns `NULL` if given object is not a layout element or if it has no predecessor branch element.

  ```
  MC_PREDECESSOR;Instance;::ofml::xoi::xOiLayoutGroup;getBranchPredecessor;$SELF
  ```

- *getBranchSuccessor(pObj(MObject)) → MObject | Void*

  Returns the topological successor branch element of given layout element.

  Returns `NULL` if given object is not a layout element or if it has no successor branch element.

  ```
  MC_SUCCESSOR;Instance;::ofml::xoi::xOiLayoutGroup;getBranchSuccessor;$SELF
  ```

- *getForkPredecessor(pObj(MObject)) → MObject | Void*

  Returns the topological predecessor fork element in the branch of given layout element.

  Returns `NULL` if given object is not a layout element or if it has no predecessor fork element in the branch.

  ```
  MC_PREDECESSOR_FORK;Instance;::ofml::xoi::xOiLayoutGroup;getForkPredecessor;$SELF
  ```

- *getForkSuccessor(pObj(MObject)) → MObject | Void*

  Returns the topological successor fork element in the branch of given layout element.

  Returns `NULL` if given object is not a layout element or if it has no successor fork element in the branch.

  ```
  MC_SUCCESSOR_FORK;Instance;::ofml::xoi::xOiLayoutGroup;getForkSuccessor;$SELF
  ```

- *getFirstForkElem(pBranch(Int)) → MObject | Void*

  Returns the first fork element from the topolopical element list of given branch.

  Returns `NULL` if there is no branch with given number or if the branch does not contain fork elements.

  ```
  MC_FIRST_FORK_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;getFirstForkElem;1
  ```

- *getLastForkElem(pBranch(Int)) → MObject | Void*

  Returns the last fork element from the topolopical element list of given branch.

  Returns `NULL` if there is no branch with given number or if the branch does not contain fork elements.

  ```
  MC_LAST_FORK_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;getLastForkElem;1
  ```

- *getFirstCatElem(pBranch(Int), pCat(Symbol)) → MObject | Void*

  Returns the first element from the topolopical element list of given branch belonging to specified element category.

  Returns `NULL` if there is no branch with given number or if the branch does not contain elements belonging to specified category.

  ```
  MC_FIRST_SEAT_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;getFirstCatElem;1,@Seat
  ```

- *getLastCatElem(pBranch(Int), pCat(Symbol)) → MObject | Void*

  Returns the last element from the topolopical element list of given branch belonging to specified element category.

  Returns `NULL` if there is no branch with given number or if the branch does not contain elements belonging to specified category.

  ```
  MC_LAST_SEAT_IN_ROOT_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;getLastCatElem;1,@Seat
  ```

9.4 Other methods

- *startNewBranch() → Void*

  Forces the creation of a new branch when the next layout element is added.

  Then, if the predecessor of the new layout element only has one neighbor it will not be marked as a fork element.

  ```
  MC_START_NEW_BRANCH;Instance;::ofml::xoi::xOiLayoutGroup;startNewBranch;
  ```

- *replaceElement(pEl(MObject), pPID(Symbol), pArticle(String), pOptions(Any), pPredecessorAttPt(Symbol), pElAttPt(Symbol), pSuccessorAttPt(Symbol)) → MObject | Void*

  Replaces given group element by an instance of specified article[13].

  If parameter *pEl* is `NULL` and the group contains only a single topological element, this element will be replaced.

  Parameter *pPID* specifies the OFML program ID.
  Parameter *pArticle* specifies the base article number.

[13] This method can be used in situations where there is no suitable property for an action of type *PropEdit* (or *PropChange*) in order to change the article of the active object.

Parameter *pOptions* either is `NULL` or a Vector of additional options for article creation:

1. Varcode type (Symbol: @VarCode, @OFMLVarCode, @Final)
2. Varcode (String)
3. PropValues (Vector of [key, value] pairs)
4. PropStates (Vector of [key, state] pairs)

All elements in this Vector may be `NULL`. Elements 3 and 4 may be omitted.

Parameter *pPredecessorAttPt* specifies the attach point of the predecessor element to be used to position the new element.
Parameter *pElAttPt* specifies the attach point of the new element to be used to connect with the successor element.
Parameter *pSuccessorAttPt* specifies the attach point of the successor element to be used to connect with the new element.

Returns `NULL` if the new element could not be created, otherwise the reference to the new article instance.

```
MC_CREATE_XY;Instance;::ofml::xoi::xOiLayoutGroup;replaceElement;\
$SELF,@foo_bar,"XY",NULL,@AP_R,@AP_R,@AP_L
```

Notes:

- It is the responsibility of the client to specify a replacement article that has the same dimensions as the replaced element!
  I.e., in contrast to *xOiJointPlGroup* no elements will be re-positioned if the new element has another width than the replaced one, even if dimension change of layout elements is enabled according to option *@InsertMode* in the control data table [an0601]. (Maybe/probably, in the future this behavior will be changed/enhanced.)
- If the interactor with the *MethodCall* action is bound to a group element (not to the planning group instance), after the 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!

- *flipElement(pEl(MObject), pPredecessorCon(Any), pSuccessorCon(Any))* → *Int*

  Switches the front/back orientation of given layout element.

  The operation is not allowed/possible if given element has more than two neighbors!

  Parameters *pPredecessorCon* and *pSuccessorCon* can be used to specify the attach points to be used to connect the flipped element with its predecessor resp. successor element if the element is flipped from its original orientation: The parameters either are `NULL` or a *Vector*

  1. attach point of predecessor (Symbol)
  2. attach point of element (Symbol)

  resp.

  1. attach point of element (Symbol)
  2. attach point of successor (Symbol)

  where the Vector elements may be `NULL`.

  If a connection information is missing (`NULL`), it is be determined as follows:

  - The attach points of predecessor and successor elements used to connect with the flipped element stay the same.
  - The attach points of the flipped element used to connect to its predecessor resp. successor elements will be switched.
    Note:
    If the flipped element has no predecessor, *pSuccessorCon* must not be `NULL` and has to specify an attach point in its first element!
    Vice versa, if the flipped element has no successor, *pPredecessorCon* must not be `NULL` and has to specify an attach point in its second element!

If the element is flipped back to its original orientation, parameters *pPredecessorCon* and *pSuccessorCon* will be ignored and the original attach point connections will be restored.

If given layout element is the first branch element and the branch of the element is not the root branch and if the element is a neighbor of the element of the parent branch where the element branch sets off, that element of the parent branch is used as the predecessor.

If given layout element is the last branch element and if it is a node where a child branch sets off, the neighbor element of that branch is used as the successor.

Returns 1 (true) if the operation was successful, otherwise 0 (false).

Important note:
If a new neighbor is up to be added to a flipped element as successor or predeccessor (i.e., if the flipped element currently has only a predeccessor resp. only a successor), in order to get the correct original orientation for the new element, the flipped element temporarily has to be switched back to its original orientation, and to be flipped again after the new neighbor element was added. (Whether an element is flipped can be determined by method *isFlippedElement()*, see below.)

```
MC_FLIP_ME;Instance;::ofml::xoi::xOiLayoutGroup;flipElement;$SELF,NULL,NULL
```

- *isFlippedElement(pEl(MObject))* → *Int*

  Returns 1 (true) if given layout element currently is flipped with respect to its original front/back orientation.

  This refers to the flip operation performed by method *flipElement()* (see above).

```
MC_AM_I_FLIPPED;Instance;::ofml::xoi::xOiLayoutGroup;isFlippedElement;$SELF
```

- *swapNeighbors(pEl1(MObject), pEl2(MObject), ...)* → *Int*

  Swaps the given two neighbor elements (belonging to the same branch).

  The elements change their geometric position as well as their position in the topological element list of the branch (incl. adaptation of connection information of the affected elements).

  The method has no effect in the following situations:
  - One of the elements is not a layout element.
  - One of the elements has more than two neighbors.
  - The elements do not belong to the same branch.
  - The elements aren't neighbors.
  - One of the elements or the predecessor of the first element (if any) or the successor of the second element (if any) currently is flipped.

    (Consider to temporarily switch affected element back to its original orientation, and to flip it again after the swap operation.)

  Returns 1 (true) if the operation was successful, otherwise 0 (false).

  Note:
  Current implementation expects, that the given elements as well as predecessor and successor (if any) use the same pairs of attach points to connect to each other. If this is not the case, correct functioning cannot be guaranteed. (Additional parameters may be added in the future, to handle other cases, too.)

```
MC_SWAP_ME_AND_NEIGHBOR_R;Instance;::ofml::xoi::xOiLayoutGroup;swapNeighbors;\
$SELF,methodCall("AC_call_NEIGHBOR_R")
```

- *canSwapNeighbors(pEl1(MObject), pEl2(MObject), ...)* → *Int*

  Returns 1 (true) if given two neighbor elements can be swapped.

  Checks the conditions mentioned above for method *swapNeighbors()*.

```
MC_CAN_SWAP_ME_AND_R;Instance;::ofml::xoi::xOiLayoutGroup;canSwapNeighbors;\
$SELF,methodCall("AC_call_NEIGHBOR_R")
```

- *setAdjustSide4Insertion(pAdjustSide(Symbol |MObject), ...) → Void*

  Specifies the neighbors of the reference object which have to be moved due to upcoming insertion of a new element between the reference object and its current neighbor.

  The parameter either specifies the attach point of the reference object to which the relevant immediate neighbor is attached, or specifies the immediate neighbor itself.

  If this method is not called before the upcoming insertion, the current active attach point of the reference object is used (if any) to determine the neighbors to be adjusted (if there is a neighbor attached to that attach point).

  If an additional optional parameter is given it specifies the attach point (*Symbol*) of the upcoming new neighbor to be used for re-positioning the neighbors at *pAdjustSide*. If the parameter is not given or is not a *Symbol*, the same attach point is used that determines the neighbors to be adjusted (see above), if the upcoming new neighbor has such an attach point.

  ```
  MC_SET_ADJUST_SIDE_4_INSERT;Instance;::ofml::xoi::xOiLayoutGroup;\
  setAdjustSide4Insertion;@AP_R
  ```

- *setAdjustSide4Removal2(pAdjustSide(Symbol |MObject), ...) → Void*

  Specifies the neighbors which have to be moved due to upcoming removal of a layout element between two neigbors.

  The parameter either specifies the attach point of the element up to be removed to which the relevant immediate neighbor is attached, or specifies the immediate neighbor itself.

  Note: Removal of an inner element is allowed/possible only if it has exactly two neighbors!

  The neighbor at the other side as *pAdjustSide* will be used as the reference object for re-positioning the neighbors at *pAdjustSide*.

  If an additional optional parameter is given it specifies the attach point (*Symbol*) of the reference object to be used for re-positioning the neighbors at *pAdjustSide*. If the parameter is not given or is not a *Symbol*, the attach point of reference object is used that was used to connect it with the element up to be removed.

  Note:
  If the neighbor elements or the element up to be removed currently are flipped (see method *flipElement()* above) the affected elements temporarily should be switched back to their original orientation, and be flipped again after the removal[14]. (Whether an element is flipped can be determined by method *isFlippedElement()*, see above.)

  ```
  MC_SET_ADJUST_SIDE_4_REMOVE;Instance;::ofml::xoi::xOiLayoutGroup;\
  setAdjustSide4Removal2;@AP_R
  ```

9.5 Additional elements

- *hasAdditionalElements() → Int*

  Returns True (1) if the group contains additional elements.

  ```
  MC_HAS_OTHER_ELEMS;Instance;::ofml::xoi::xOiLayoutGroup;hasAdditionalElements;
  ```

- *isAdditionalElement(pObj(MObject)) → Int*

  Returns True (1) if given instance is contained in the list of additional elements.

  ```
  MC_IS_ADDITIONAL_EL;Instance;::ofml::xoi::xOiLayoutGroup;isAdditionalElement;$SELF
  ```

[14] in case of the neighbors

10 Class *xOiTabularPlGroup*

Base class for planning groups with a tabular layout.

Instances may have an horizontal orientation (XZ plane) or a vertical orientation (XY plane).

Elements aligned parallel to X axis form *rows*, elements aligned parallel to Y or Z axes are called *columns* (or *lines* with an horizontal layout). The crossing of a row and a column forms a *field*.

Fields have an *address* consisting of the index of the corresponding column and the index of the corresponding row, where the index numbering starts with zero. Two fields, where the absolute index difference in both directions is 0 or 1, are called *neighbor fields*. Two neighbor fields, having the same index in one direction, are called *straight neighbor fields*.

Each column has a defined *width* and each row has a defined *depth* with the horizontal orientation resp. a defined *height* with the vertical orientation, but the group may be configured so that all columns have the same (uniform) width and all rows have the same depth resp. height.

Project specific subclasses must not be derived from *xOiTabularPlGroup* but have to be derived from dedicated subclasses according to required orientation and row count direction:

xOiTabularPlGroupVert
: Realizes a tabular planning group with a vertical orientation.

xOiTabularPlGroupHForth
: Realizes a tabular planning group with a horizontal orientation where rows are added in positive Z direction, i.e., forwards (back to front).

xOiTabularPlGroupHBack
: Realizes a tabular planning group with a horizontal orientation where rows are added in negative Z direction, i.e., backwards (front to back).

There are two supported types of group elements:

- Elements defining the *layout* of the group, also called *field elements*.
- Additional, *other* (non-layout) elements.

  Typically, they are created based on the current layout (defined by the layout elements), e.g. frames for sofa arrangements, links between field elements and alike.

  Creating such elements requires special OFML programming, i.e., this cannot be done in OAP data alone. Therefore, methods related to non-layout elements are not covered in this document.

A field element may occupy more than one field, in which case it is called a *multi-field element*. The field, that defines the position of a multi-field element is called its *root field*.

Note: Some operations (methods) manipulating the tabular structure are not allowed/possible if multi-field elements would be affected. This will be stated below for each affected method. Methods *hasMultiFieldElements()* and *isStraightLine()* (see subsection 10.2 below) can be used for according pre-checks.

Two single-field elements are called *neighbors* if they are located in neighbor fields (see definition above).

The *neighbors of a multi-field element* are determined as follows:
For all fields occupied by the element, the first field in the requested search direction is determined, which is not occupied by the multi-field element. If this field is occupied by another field element (or placeholder), that element (or placeholder) is a neighbor in the requested search direction.
The neighbor relative to the root field of a multi-field element is called its *main neighbor*.

The width and the depth resp. height of a field element can be smaller than the width resp. depth/height defined for the according column resp. row of the field. Accordingly, various alignment strategies are supported.

A field may be empty. Then, it can be filled with a *placeholder* (not being a field element).

When a field element is deleted, by default, empty head and back columns resp. rows after the removal of the element will be removed from the internal data structures. This can be avoided by using method *deleteFieldElement2()* with value 1 (true) for parameter *pKeepFields* (see subsection 10.3 below).

Some aspects of the behaviour may be controlled by specifying according options in control data table `tabularplgroup` [an0601].

10.1 General settings and information

- *setMaxColumns(pCount(Int))* → *Void*

  Assigns new maximum allowed count of columns. (See also option *@MaxColumns* in [an0601].)

  Value 0 denotes no restriction.

  The assignment of the given count (greater than 0) is rejected if it is lower than the current count of columns.

  ```
  MC_SET_MAX_COLUMNS_6;Instance;::ofml::xoi::xOiTabularPlGroup;setMaxColumns;6
  ```

- *getMaxColumns()* → *Int*

  Returns currently defined maximum allowed count of columns.

  (Value 0 denotes no restriction.)

  ```
  MC_GET_MAX_COLUMNS;Instance;::ofml::xoi::xOiTabularPlGroup;getMaxColumns;
  ```

- *setMaxRows(pCount(Int))* → *Void*

  Assigns new maximum allowed count of rows. (See also option *@MaxRows* in [an0601].)

  Value 0 denotes no restriction.

  The assignment of the given count (greater than 0) is rejected if it is lower than the current count of rows.

  ```
  MC_SET_MAX_COLUMNS_5;Instance;::ofml::xoi::xOiTabularPlGroup;setMaxRows;5
  ```

- *getMaxRows()* → *Int*

  Returns currently defined maximum allowed count of rows.

  (Value 0 denotes no restriction.)

  ```
  MC_GET_MAX_COLUMNS;Instance;::ofml::xoi::xOiTabularPlGroup;getMaxRows;
  ```

- *setUniformFieldSize(pColumnWidth(Float | Void), pRowSize(Float | Void))* → *Void*

  Assigns new dimension(s) for uniformly sized fields.

  Parameter *pColumnWidth* either is `NULL` or specifies the new column width. The parameter will be ignored, if columns do not have a uniform width, according to option *@UniformColumnWidth*. (See also option *@ColumnWidth*.)

  Parameter *pRowSize* either is `NULL` or specifies the new row height resp. depth. The parameter will be ignored, if rows do not have a uniform size, according to options *@UniformRowSize/Height/Depth*. (See also options *@RowSize/Height/Depth*.)

  Note: if a size change becomes effective, field elements will be re-positioned accordingly.

  ```
  MC_SET_UNIFORM_FIELD_SIZE;Instance;::ofml::xoi::xOiTabularPlGroup;\
  setUniformFieldSize;NULL,2.4
  ```

- *getUniformColumnWidth() → Float | Void*

  Returns the current width for uniformly sized fields.

  Return value is `NULL` if uniform column width is not enabled according to option *@UniformColumnWidth*.

  ```
  MC_GET_UNIFORM_COLUMN_WIDTH;Instance;::ofml::xoi::xOiTabularPlGroup;\
  getUniformColumnWidth;
  ```

- *getUniformRowSize() → Float | Void*

  Returns the current height resp. depth for uniformly sized fields.

  Return value is `NULL` if uniform row size is not enabled according to options *@UniformRowSize/Height/Depth*.

  ```
  MC_GET_UNIFORM_ROW_SIZE;Instance;::ofml::xoi::xOiTabularPlGroup;getUniformRowSize;
  ```

- *getColumnWidth(pIdx(Int)) → Float | Void*

  Returns the width of the column with specified index.

  Return value is `NULL`, if there is no column with specified index.

  ```
  MC_GET_COLUMN_2_WIDTH;Instance;::ofml::xoi::xOiTabularPlGroup;getColumnWidth;2
  ```

- *getRowSize(pIdx(Int)) → Float | Void*

  Returns the height resp. depth of the row with specified index.

  Return value is `NULL`, if there is no row with specified index.

  ```
  MC_GET_ROW_3_HEIGHT;Instance;::ofml::xoi::xOiTabularPlGroup;getRowSize;3
  ```

- *setPlaceholderCreationOn(pValue(Int)) → Void*

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

  Can be used to dynamically change the value for option *@EmptyFieldsPlaceholder*. (See also information on accompanying options.)

  ```
  MC_SET_PLACEHOLDER_CREATION_ON;Instance;::ofml::xoi::xOiTabularPlGroup;\
  setPlaceholderCreationOn;1
  ```

- *getPlaceholderCreationOn()) → Int*

  Returns the current value of option *@EmptyFieldsPlaceholder*.

  ```
  MC_IS_PLACEHOLDER_CREATION_ON;Instance;::ofml::xoi::xOiTabularPlGroup;\
  getPlaceholderCreationOn;
  ```

10.2 Information on fields and layout structure

- *getColumnCount()) → Int*

  Returns current count of columns.

  ```
  MC_COLUMN_COUNT;Instance;::ofml::xoi::xOiTabularPlGroup;getColumnCount;
  ```

- *getRowCount()) → Int*

  Returns current count of rows.

  ```
  MC_ROW_COUNT;Instance;::ofml::xoi::xOiTabularPlGroup;getRowCount;
  ```

- *isLayoutEmpty()) → Int*

  Returns 1 (true) if the group does not contain any field element.

  ```
  MC_IS_GROUP_EMPTY;Instance;::ofml::xoi::xOiTabularPlGroup;isLayoutEmpty;
  ```

- *firstFieldElement()) → MObject | Void*

  Returns the first entry from the list of current field elements (or an instance of *Void* if the list currently is empty).

  ```
  MC_FIRST_FIELD_EL;Instance;::ofml::xoi::xOiTabularPlGroup;firstFieldElement;
  ```

- *lastFieldElement()) → MObject | Void*

  Returns the last entry from the list of current field elements (or an instance of *Void* if the list currently is empty).

  ```
  MC_LAST_FIELD_EL;Instance;::ofml::xoi::xOiTabularPlGroup;lastFieldElement;
  ```

- *isFieldElement(pObj(MObject)) → Int*

  Returns 1 (true) if given instance is a field element.

  ```
  MC_AM_A_FIELD_EL;Instance;::ofml::xoi::xOiTabularPlGroup;isFieldElement;$SELF
  ```

- *isMultiFieldElement(pObj(MObject)) → Int*

  Returns 1 (true) if given instance is a field element and occupies multiple fields.

  ```
  MC_AM_A_MULTI_FIELD_EL;Instance;::ofml::xoi::xOiTabularPlGroup;isMultiFieldElement;\
  $SELF
  ```

- *getFieldOf(pObj(MObject)) → Int[2] | Void*

  Returns the field address of given field element or placeholder.

  Returns `NULL` if given object is not a field element and no placeholder, otherwise the root field, i.e. the field that was used for the position of the element resp. placeholder.

  ```
  MC_MY_FIELD_ADDRESS;Instance;::ofml::xoi::xOiTabularPlGroup;getFieldOf;$SELF
  ```

- *getColumnOf(pObj(MObject)) → Int | Void*

  Returns the index of the column of given field element or placeholder.

  Returns `NULL` if given object is not a field element and no placeholder, otherwise the column of the root field, i.e. the field that was used for the position of the element resp. placeholder.

  ```
  MC_MY_COLUMN_IDX;Instance;::ofml::xoi::xOiTabularPlGroup;getColumnOf;$SELF
  ```

- *getRowOf(pObj(MObject)) → Int | Void*

  Returns the index of the column of given field element or placeholder.

  Returns `NULL` if given object is not a field element and no placeholder, otherwise the column of the root field, i.e. the field that was used for the position of the element resp. placeholder.

  ```
  MC_MY_ROW_IDX;Instance;::ofml::xoi::xOiTabularPlGroup;getRowOf;$SELF
  ```

- *hasMultiFieldElements(pObjs(MObject | Void)) → Int*

  Returns 1 (true) if there is at least one field element occupying multiple fields.

  If parameter *pObjs* is not `NULL` but specifies a *List* or *Vector* of field elements (or placeholders), only those elements will be checked.

  ```
  MC_HAS_MULTIFIELD_ELEMENTS;Instance;::ofml::xoi::xOiTabularPlGroup;\
  hasMultiFieldElements;NULL
  ```

- *isStraightLine(pLineType(Symbol), pLineIdx(Int)) → Int*

  Returns 1 (true) if none of the elements in given line occupies fields of another line (i.e., if they form a straight line).

  Possible types in parameter *pLineType* are `@Column` and `@Row`.

  ```
  MC_IS_COLUMN_1_STRAIGHT;Instance;::ofml::xoi::xOiTabularPlGroup;\
  isStraightLine;@Column,1
  ```

- *getNeighbor(pObj(MObject | Int[2]), pDir(Symbol), ...) → MObject | Void*

  Returns the neighbor element of given field element in given direction.

  In parameter *pObj*, instead of a field element, the address of a field may be passed.

  The possible direction values (parameter *pDir*) are:

  `@N` (or `@North`, `@Up` and `@Back`)

  `@E` (or `@East`, `@R` and `@Right`)

  `@S` (or `@South`, `@Down` and `@Forth`)

  `@W` (or `@West`, `@L` and `@Left`)

  and the diagonal directions: `@NE`, `@SE`, `@SW`, `@NW`.

  Additional optional parameter of type *Int* specifies whether placeholders in empty fields have to be considered (1) or not (0, default).

  Returns `NULL` if given direction is not valid or if given instance is not a field element (resp. a placeholder) or if there is no neighbor element (resp. placeholder) in given given direction.

  If the given element occupies more than one field, the neighbor relativ to the root field will be returned.

  ```
  MC_NEIGHBOR_R;Instance;::ofml::xoi::xOiTabularPlGroup;getNeighbor;$SELF,@R
  ```

10.3 Manipulating the layout structure

- *addNeighbor(pRefField(MObject | Int[2]), pDir(Symbol), pArticle(Any), pAlignment(Any)) → MObject | Void*

  Creates a new field next to specified reference field in given direction (if it does not exist yet) and fills it with specified article.

  The reference field either is specified by its address or by the according field element. If an element is given it must not occupy more than one field.

  For possible direction values (parameter *pDir*) see method *getNeighbor()* above.

  Parameter *pArticle* either is a specification according to option *@DefaultArticle* or is `NULL`, in which case an article is created as specified in option *@DefaultArticle*.

  Parameter pAlignment is not supported yet. (New element is aligned according to corresponding options.)

  The method has no effect if the maximum allowed count of columns or rows (see options *@MaxColumns* and *@MaxRows*) has already been reached.

If the direction denotes a new element before first column resp. row, the indices of all existing columns resp. rows will be incremented and new column resp. row (for the new element) gets index 0. The position of the new element gets a negative coordinate in the according dimension.
This scenario is not allowed/possible if the group contains at least one element occupying more than one field.
Also note: do not use this scenario if the fields do not have the same size.

Returns `NULL` if the operation is not allowed (see above) or in case of invalid parameters or if the neighbor field violates the maximum allowed count of columns or rows or if the article could not be created. Otherwise returns the reference to the new element.

```
MC_ADD_NEIGHBOR_R;Instance;::ofml::xoi::xOiTabularPlGroup;addNeighbor;\
$SELF,@R,NULL,NULL
```

- *addColumn2(pArtSpecType(Symbol), pArticle(Any), pDir(Symbol), pAlignment(Any), pWidth(Float), ...) → Void*

  Creates a new column and fills it with specified article(s).

  Parameter *pArtSpecType* specifies the type of article creation for the new fields and defines the expected type of parameter *pArticle*:

  `@Default`
  : All fields will be filled by articles according to option *@DefaultArticle*.
    Parameter pArticle is ignored (should be `NULL`).

  `@Unique`
  : All fields will be filled by identical articles.
    Parameter *pArticle* is a specification according to option *@DefaultArticle*.

  `@Individual`
  : The fields will be filled individually.
    Parameter *pArticle* is a *Vector* whose elements either are `NULL`, if the according field should be empty, or are a specification according to option *@DefaultArticle*.

  `@Empty`
  : All fields are empty (possibly filled with placeholders assording to option *@EmptyFieldsPlaceholder*).
    Parameter *pArticle* specifies the number (*Int*) of fields to be created in the column.

  Parameter *pDir* specifies the direction (side) where the new column is added:

  `@Last`
  : new column is added to the right of current last column

  `@First`
  : new column is added to the left of current first column

  Parameter *pAlignment* is not supported yet. (New elements are aligned according to corresponding options.)

  Parameter *pWidth* specifies the width of new column:
  If all columns have the same width (see option *@UniformColumnWidth*), the parameter is ignored. Otherwise, if necessary, it is used to extend the Vector of widths for non-uniformly sized columns (see option *@ColumnWidths*):

  – In case of direction `@First`, the parameter value will be appended to the Vector as the new first elememt.
  – In case of direction `@Last`, the parameter value will be appended to the Vector as the new last elememt if the current Vector size does not contain an element for the new column to be added.

  In case of direction `@First` the indices of all existing columns will be incremented and the new column gets index 0. The position of the elements in the new column get a negative coordinate in X dimension.

The method has no effect if the maximum allowed count of columns (see option *@MaxColumns*) has already been reached.

```
MC_ADD_COLUMN_R;Instance;::ofml::xoi::xOiTabularPlGroup;addColumn2;\
@Default,NULL,@Last,NULL,NULL
```

- *addRow3(pArtSpecType(Symbol), pArticle(Any), pDir(Symbol), pAlignment(Any), pSize(Float), ...) → Void*

  Creates a new row and fills it with specified article(s).

  Parameter *pArtSpecType* specifies the type of article creation for the new fields and defines the expected type of parameter *pArticle*. For the possible values see same parameter of method *addColumn2()* above.

  Parameter *pDir* specifies the direction (side) where the new row is added:

  `@Last`
  : new row is added after current last row

  `@First`
  : new row is added before current first row

  Parameter *pAlignment* is not supported yet. (New elements are aligned according to corresponding options.)

  Parameter *pSize* specifies the width of new row:
  If all rows have the same size (see option *@UniformRowSize*), the parameter is ignored. Otherwise, if necessary, it is used to extend the Vector of sizes for non-uniformly sized rows (see option *@RowSizes*):

  – In case of direction `@First`, the parameter value will be appended to the Vector as the new first elememt.
  – In case of direction `@Last`, the parameter value will be appended to the Vector as the new last elememt if the current Vector size does not contain an element for the new row to be added.

  In case of direction `@First` the indices of all existing rows will be incremented and the new row gets index 0. The position of the elements in the new row get a negative coordinate in according dimension.

  The method has no effect if the maximum allowed count of rows (see option *@MaxRows*) has already been reached.

  ```
  MC_ADD_EMPTY_ROW_F_SIZE3;Instance;::ofml::xoi::xOiTabularPlGroup;addRow3;\
  @Empty,3,@First,NULL,NULL
  ```

- *deleteFieldElement2(pField(MObject | Int[2]), pKeepFields(Int), ...) → Void*

  Deletes the element from given field.

  The field either is specified by its address or by the according field element.

  Parameter *pKeepFields* specifies whether any fields have to be kept (1) or not (0). If not, empty head and back columns resp. rows after the removal of the element will be removed from the internal data structures.

  Note:
  If the interactor with the *MethodCall* action is bound to a group element (not to the planning group instance), after the 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!

  ```
  MC_DELETE_ME;Instance;::ofml::xoi::xOiTabularPlGroup;deleteFieldElement2;$SELF,0
  ```

- *deleteColumn(pColumn(Int), pKeepFields(Int)) → Void*

  Deletes all elements from the column specified by given index.

  If `NULL` is passed for parameter *pColumn* the last column will be deleted.

  Parameter *pKeepFields* specifies whether the fields have to be kept (1) or not (0). If not, the complete column will be removed, i.e. the columns with higher indexes will be moved (shifted) to fill the place of the removed column (including repositioning of the contained elements).

  Note:
  Currently, this implementaion does not support shifting columns with different widths, i.e., in this case the method only can be used to delete the last column!

  The operation is not allowed/possible if the specified column contains at least one element occupying fields of another column. (This can be checked in advance using method *isStraightLine()*, see above in section 10.2.)

  Consider usage of method *deleteLine()* (see below) providing more options to control the behaviour.

  ```
  MC_DELETE_LAST_COLUMN;Instance;::ofml::xoi::xOiTabularPlGroup;deleteColumn;NULL,0
  ```

- *deleteRow(pRow(Int), pKeepFields(Int)) → Void*

  Deletes all elements from the row specified by given index.

  If `NULL` is passed for parameter *pRow* the last row will be deleted.

  Parameter *pKeepFields* specifies whether the fields have to be kept (1) or not (0). If not, the complete row will be removed, i.e. the rows with higher indexes will be moved (shifted) to fill the place of the removed row (including repositioning of the contained elements).

  Note:
  Currently, this implementaion does not support shifting rows with different sizes, i.e., in this case the method only can be used to delete the last row!

  The operation is not allowed/possible if the specified row contains at least one element occupying fields of another row. (This can be checked in advance using method *isStraightLine()*, see above in section 10.2.)

  Consider usage of method *deleteLine()* (see below) providing more options to control the behaviour.

  ```
  MC_DELETE_FIRST_ROW;Instance;::ofml::xoi::xOiTabularPlGroup;deleteRow;0,0
  ```

- *deleteLine(pType(Symbol), pIdx(Int), pKeepFields(Int), pShiftDir(Symbol), ...) → Void*

  Deletes all elements from the line specified by given type and index.

  Parameter *pType* either is `@Row` or `@Column`.

  If `NULL` is passed for parameter *pIdx* the last row resp. column will be deleted.

  Parameter *pKeepFields* specifies whether the fields have to be kept (1) or not (0). If not, the complete row resp. column will be removed, i.e. the other rows resp. columns will be moved (shifted) to fill the place of the removed line.

  The shift direction is controlled by parameter *pShiftDir*:

  `@Left`
  : lines with higher indices will be shifted (by 1) to line with lower index

  `@Right`
  : lines with lower indices will be shifted (by 1) to line with higher index

  Note:
  Currently, this implementaion does not support shifting lines with different sizes, i.e., in this case the method only can be used to delete the last line!

The operation is not allowed/possible if the specified line contains at least one element occupying fields of another line. (This can be checked in advance using method *isStraightLine()*, see above in section 10.2.)

```
MC_DELETE_LAST_COLUMN_SHIFT_R;Instance;::ofml::xoi::xOiTabularPlGroup;\
deleteLine;@Column,NULL,0,@Right
```

- *resize(pColumns(Int), pRows(Int), pArticle(Any), pAlignment(Any)) → Void*

  Resizes the group according to given numbers for the new count of columns resp. rows.

  If one of the counts is greater than the current count new fields will be created filling it with the specified article:
  Parameter *pArticle* either is a specification according to option *@DefaultArticle* or is `NULL`, in which case articles are created as specified in option *@DefaultArticle*.

  If there is specified a maximum allowed count of columns resp. rows (see options *@MaxColumns* resp. *@MaxRows*), these counts will be used for resizing if the values passed in parameters *pColumns* or *pRows* are greater.

  Note:
  This method supports a resize operation only in positive (index count) direction (both for columns and rows).

  Parameter pAlignment is not supported yet. (New elements are aligned according to corresponding options.)

  The operation is not allowed/possible, if the group contains at least one element occupying more than one field. (This can be checked in advance using method *hasMultiFieldElements()*, see above in section 10.2.)

  ```
  MC_RESIZE_4X4;Instance;::ofml::xoi::xOiTabularPlGroup;resize;4,4,NULL,NULL
  ```

- *shiftColumns(pDir(Symbol), pCircular(Any)) → Void*

  Shifts all columns by one column in specified direction.

  The possible direction values (parameter *pDir*) are:

  `@Left`
  : shifting towards columns with lower indexes

  `@Right`
  : shifting towards columns with higher indexes

  Parameter *pCircular* currently is unused. The method performs a circular shift, i.e. the column shifted out replaces the (empty) column on the other side.

  The operation is not allowed/possible if the group contains at least one element occupying fields of multiple columns!

  Furthermore, in case of non-uniformly sized columns it is required that the count of assigned column sizes (see option *@ColumnWidths*) is equal to the current count of columns! (The elements in that *Vector* will be shifted accordingly.)

  ```
  MC_SHIFT_COLUMNS_R;Instance;::ofml::xoi::xOiTabularPlGroup;shiftColumns;@Right,NULL
  ```

- *shiftRows(pDir(Symbol), pCircular(Any)) → Void*

  Shifts all rows by one row in specified direction.

  The possible direction values (parameter *pDir*) are:

  `@Down`
  : shifting towards rows with lower indexes

  `@Up`
  : shifting towards rows with higher indexes

Parameter *pCircular* currently is unused. The method performs a circular shift, i.e. the row shifted out replaces the (empty) row on the other side.

The operation is not allowed/possible if the group contains at least one element occupying fields of multiple rows!

Furthermore, in case of non-uniformly sized rows it is required that the count of assigned row sizes (see option *@RowSizes*) is equal to the current count of rows! (The elements in that *Vector* will be shifted accordingly.)

```
MC_SHIFT_ROWS_UP;Instance;::ofml::xoi::xOiTabularPlGroup;shiftRows;@Up,NULL
```

- *replaceField(pRepl(MObject |Int[2]), pArticle(Any), pAlignment(Any)) → MObject |Void*

  Replaces given field (element) by an instance of specified article.

  In parameter *pRepl*, instead of a field element, the address of a field or a placeholder instance may be passed.

  The replacement article can be specified in parameter pArticle in the following ways:

  – `NULL`:
    Creates article according to option *@DefaultArticle*.
  – Field address (*Int[2]*):
    Clones article from specified field[15].
  – Explicit article (Any[]):
    Parameter *pArticle* is a specification according to option *@DefaultArticle*.

  Parameter *pAlignment* is not supported yet. (New element is aligned according to corresponding options.)

  In case of invalid parameters, the method has no effect, i.e. the element specified by parameter *pRepl* (if any) will *not* be removed.

  It is the responsibility of the client to specify a replacement article that fits into the field resp., in case of a multi-field element, that relevant neighbor fields are empty.

  If a new element (successfully) has been created, it gets the name of the replaced field element (if any) and return value is the reference to this new instance, otherwise return value is `NULL`.

  Note:
  If the interactor with the *MethodCall* action is bound to a group element (not to the planning group instance), after the action no further actions may follow, which again need a target object or an object definition as a parameter, because after the (successful) action the original active object no longer exists!

  ```
  MC_REPLACE_ME_BY_DEFAULT;Instance;::ofml::xoi::xOiTabularPlGroup;replaceField;\
  $SELF,NULL,NULL
  ```

- *moveFieldElement(pFieldEl(MObject), pDestType(Symbol), pDestination(Any), ...) → Int[2] |Void*

  Moves given field element to a new destination (field).

  Currently, this operation can be performed only if all columns have the same width (see option *@UniformColumnWidth*) and if all rows have the same depth resp. height (see option *@UniformRowSize*).

[15] If that field is empty the field specified by *pRepl* becomes empty, too.

The destination (parameter *pDestination*) may be given in two ways. Which way is used is specified in parameter *pDestType*:

`@Address`
: The destination is determined by a field address (*Int[2]*).

`@DirNumber`
: The destination is determined by the direction of movement and a number of fields by which to move the element in that direction.
The according value in parameter *pDestination* is a Vector of

1. direction (*Symbol*), for possible values see method *getNeighbor()*
2. number of fields (*Int*)

The operation will be rejected if the element at the new destination would occupy fields which are already occupied by other field elements.

If an additional optional parameter is given, it specifies whether the element may occupy non-existing fields at the new destination. If this parameter has value 1 (yes/true) and the element would occupy non-existing fields at the new destination, according rows resp. columns will be added to the table structure.
Default is 0 (no/false), i.e., the operation will be rejected if the element would occupy non-existing fields at the new destination.

If the operation is rejected, return value is `NULL`, otherwise the new address of the element.

Destination type *@DirNumber* can be used to implement an action bound to an interactor with symbol types *Pos2Left* and the like:

```
MC_MOVE_ME_LEFT;Instance;::ofml::xoi::xOiTabularPlGroup;moveFieldElement;\
$SELF,@DirNumber,[@L,1],1
```

11 Class *xOiCustomPlGroup*

This base class supports the realization of groups with specific (custom) planning logics.

In contrast to the other planning group classes described in the previous sections, this class does not implement specific planning logics, but supports general concepts known from those classes.
Specific (custom) logics have to be implemented in derived classes.

The following general concepts are supported:

- Controlling aspects of the behavior by means of options in a control data table, here `customplgroup.csv`. In particular, this refers to the generation of common properties. For all options see Application Note [an0601].
- Elements of the group may be assigned to self-defined categories, see section 7.2.
- A distinction can be made between 2 types of group elements:
  - Elements defining the actual *layout* of the group.
    Methods regarding layout elements are described in 11.1.
  - Additional, *other* (non-layout) elements.
    Typically, they are created based on the current layout (defined by the layout elements), e.g. frames for sofa arrangements, links between layout elements and alike.
    Methods regarding additional elements are described in 11.2.
- Generic implementation of methods *getAddStateCode()* and *setAddStateCode()* (interface *Article*). Member variables of derived classes that are to be encoded are just specified using hook method *getAddStateMembers()*.
- Generic implementation of interface *CompositeArticle*.
- Checking for free attach points (see section 11.3).

- Recording neighbor-relationships between layout elements based on OFML *attach points* (optional).

  Two layout elements are called *neighbors* if they were placed next to each other using a pair of attach points[16].

  This functionality is enabled with (default) value 1 of option *@StoreNeighborship* (see [an0601]. Then, the class behaves like a reduced *xOiLayoutGroup* without handling for branches and fork elements. This includes repositioning of layout elements after a dimension change or after insertion between two existing elements if option *@InsertMode* has the value 1 resp. 2.

  Methods regarding neighbor-relationships are described in 11.4.

11.1 Layout elements

Note:
The order of the elements in the list of layout elements reflects the order of insertion resp. removal of the elements!

- *getLayoutElCount()* → *Int*

  Returns the current number of elements in the list of layout elements.

  ```
  MC_LAYOUT_ELCOUNT;Instance;::ofml::xoi::xOiCustomPlGroup;getLayoutElCount;
  ```

- *isLayoutEmpty()* → *Int*

  Returns True (1) if the group does not contain any layout element.

  ```
  MC_IS_LAYOUT_EMPTY;Instance;::ofml::xoi::xOiCustomPlGroup;isLayoutEmpty;
  ```

- *isLayoutElement(pObj(MObject))* → *Int*

  Returns True (1) if given instance is a layout element.

  ```
  MC_IS_LAYOUT_EL;Instance;::ofml::xoi::xOiCustomPlGroup;isLayoutElement;$SELF
  ```

- *firstLayoutElement()* → *MObject* | *Void*

  Returns the first entry from the list of current layout elements (or `NULL` if the list currently is empty).

  ```
  MC_FIRST_LAYOUT_EL;Instance;::ofml::xoi::xOiCustomPlGroup;firstLayoutElement;
  ```

- *isFirstLayoutElement(pObj(MObject))* → *Int*

  Returns True (1) if given instance is the first element in the list of current layout elements.

  ```
  MC_IS_FIRST_LAYOUT_EL;Instance;::ofml::xoi::xOiCustomPlGroup;isFirstLayoutElement;$SELF
  ```

- *lastLayoutElement()* → *MObject* | *Void*

  Returns the last created element layout element (or `NULL` if the list of layout elements currently is empty).

  ```
  MC_LAST_LAYOUT_EL;Instance;::ofml::xoi::xOiCustomPlGroup;lastLayoutElement;
  ```

- *isLastLayoutElement(pObj(MObject))* → *Int*

  Returns True (1) if given instance is the last element in the list of current layout elements.

  ```
  MC_IS_LAST_LAYOUT_EL;Instance;::ofml::xoi::xOiCustomPlGroup;isLastLayoutElement;$SELF
  ```

[16] Note: Two layout elements may be located next to each other but be no neighbors because they were not placed next to each other using a pair of attach points.

- *canBeRemovedFromLayout(pObj(MObject))* → *Int*

  Returns True (1) if given layout element can be deleted (and removed from the neighbor data structure).

  A layout element can be removed (by the definition of this implementation) if option *@StoreNeighborship* has the value 0 or if one of the following conditions is met:
  - it has no neighbors but is not the single layout element left
  - it has only 1 neighbor
  - it has 2 neighbors and the insert mode is 2 (see option *@InsertMode*)

  ```
  MC_CAN_REMOVE;Instance;::ofml::xoi::xOiCustomPlGroup;canBeRemovedFromLayout;$SELF
  ```

- *setAdjustSide4Insertion(pAdjustSide(Symbol |MObject), ...)* → *Void*

  Specifies the neighbors of the reference object which have to be moved due to upcoming insertion of a new element between the reference object and its current neighbor.

  The parameter either specifies the attach point of the reference object to which the relevant immediate neighbor is attached, or specifies the immediate neighbor itself.

  If this method is not called before the upcoming insertion, the current active attach point of the reference object is used (if any) to determine the neighbors to be adjusted (if there is a neighbor attached to that attach point).

  If an additional optional parameter is given it specifies the attach point (*Symbol*) of the upcoming new neighbor to be used for re-positioning the neighbors at *pAdjustSide*. If the parameter is not given or is not a *Symbol*, the same attach point is used that determines the neighbors to be adjusted (see above), if the upcoming new neighbor has such an attach point.

  ```
  MC_SET_ADJUST_SIDE_4_INSERT;Instance;::ofml::xoi::xOiCustomPlGroup;\
  setAdjustSide4Insertion;@AP_R
  ```

- *setAdjustSide4Removal(pAdjustSide(Symbol |MObject), ...)* → *Void*

  Specifies the neighbors which have to be moved due to upcoming removal of a layout element between two neigbors.

  The parameter either specifies the attach point of the element up to be removed to which the relevant immediate neighbor is attached, or specifies the immediate neighbor itself.

  Note: Removal of an inner element is allowed/possible only if it has exactly two neighbors!

  The neighbor at the other side as *pAdjustSide* will be used as the reference object for re-positioning the neighbors at *pAdjustSide*.

  If an additional optional parameter is given it specifies the attach point (*Symbol*) of the reference object to be used for re-positioning the neighbors at *pAdjustSide*. If the parameter is not given or is not a *Symbol*, the attach point of reference object is used that was used to connect it with the element up to be removed.

  ```
  MC_SET_ADJUST_SIDE_4_REMOVE;Instance;::ofml::xoi::xOiCustomPlGroup;\
  setAdjustSide4Removal;@AP_R
  ```

11.2 Additional elements

- *hasAdditionalElements()* → *Int*

  Returns True (1) if the group contains additional elements.

  ```
  MC_HAS_OTHER_ELEMS;Instance;::ofml::xoi::xOiCustomPlGroup;hasAdditionalElements;
  ```

- *isAdditionalElement(pObj(MObject))* → *Int*

  Returns True (1) if given instance is contained in the list of additional elements.

  ```
  MC_IS_ADDITIONAL_EL;Instance;::ofml::xoi::xOiCustomPlGroup;isAdditionalElement;$SELF
  ```

11.3 Free attach points

An attach point is *free*, if it was not used to connect with a neighbor layout element and if there is no (other) sibling or child element occupying this attach point geometrically.
The first condition applies only to layout elements and requires (default) value 1 for option *@StoreNeighborship*.

- *hasFreeAttPts2(pObj(MObject), pCheckAll(Int)) → Int*

  Returns True (1) if given instance has at least one free OFML attach point, otherwise False (0).

  Parameter *pCheckAll* specifies whether all attach points of given instance have to be checked (1) or only the additional attach points (0, i.e. ignoring the standard attach points).

  ```
  MC_HAS_FREE_ATTPTS;Instance;::ofml::xoi::xOiCustomPlGroup;hasFreeAttPts2;$SELF,0
  ```

- *isFreeAttPt(pObj(MObject), pAttPt(Symbol)) → Int*

  Returns True (1) if the specified OFML attach point (key) currently is defined for the given group element and if it is free.
  Otherwise returns False (0).

  ```
  MC_IS_ATTPT_R_FREE;Instance;::ofml::xoi::xOiCustomPlGroup;isFreeAttPt;$SELF,@ATTPT_R
  ```

11.4 Neighbors

Note:
Neighbor-relationships between layout elements are stored in an internal data structure (only) if option *@StoreNeighborship* has the (default) value 1.

- *getNeighbor(pObj(MObject), pAttPt(Symbol)) → MObject | Void*

  Returns the neighbor of given layout element at given attach point.

  Returns `NULL` if given instance is not registered in the neighbor map or if there is no neighbor at given attach point.

  ```
  MC_NEIGHBOR_R;Instance;::ofml::xoi::xOiCustomPlGroup;getNeighbor;$SELF,@ATTPT_R
  ```

  See also method *getAdjacentElement()* below.

- *hasNeighbors(pObj(MObject)) → Int*

  Returns True (1) if given instance is a layout element and if it currently has registered neighbors.

  ```
  MC_HAS_NEIGHBORS;Instance;::ofml::xoi::xOiCustomPlGroup;hasNeighbors;$SELF
  ```

- *getAdjacentElement(pObj(MObject), pAttPt(Symbol)) → MObject | Void*

  Returns the group element which occupies the specified attach point of given layout element.

  In contrast to *getNeighbor()* (see above) this method also considers elements *not* being a (registered) neighbor of given layout element, i.e. which are not explicitely connected to it but are just located beside it occupying the specified attach point geometrically.

  Returns `NULL` if there is no attach point currently defined with given key or if there is no group element occupying the specified attach point.

  ```
  MC_ADJACENT_R;Instance;::ofml::xoi::xOiCustomPlGroup;getAdjacentElement;$SELF,@ATTPT_R
  ```

A Alphabetic index of methods

B Methods mapped to object types

B.1 All article types

Note:
Includes interfaces *Base*, *Complex*, *Property* and *Article*.

B.2 GoMetaType

B.3 All planning group classes

B.4 xOiJointPlGroup

B.5 xOiLayoutGroup

B.6 xOiTabularPlGroup

B.7 xOiCustomPlGroup

C Document history

Version 1.11 (September 25, 2023)

- More details on some methods regarding element categories implemented in all planning group classes (section 7.2).
- Added description of method *setSelectability4ElemCat(()* implemented in all planning group classes (section 7.2).
- Added section 7.4 describing other methods that can be used for all planning group classes, namely *isGrandSubArticle()* and *checkElDistance()*.
- Added section covering methods regarding additional elements of class *xOiLayoutGroup* (section 9.5).
- Updated description of class *xOiTabularPlGroup* with respect to neighborsip and handling of empty head and back columns resp. rows (section 10).
- Added description of methods *isFieldElement()* and *isMultiFieldElement()* for class *xOiTabularPlGroup* (section 10.2).
- Added section for class *xOiCustomPlGroup* (section 11).

Version 1.10 (December 2, 2022)

- Updated description of method *canBeRemovedFromLayout()* for class *xOiLayoutGroup* (section 9.1).

Version 1.9 (October 24, 2022)

- More details on the various implementations of *getWidth()*, *getHeight()* and *getDepth()*, possibly hindering the usage in OAP projects (section 3).
- Added description of methods *getPropState2()* and *checkPropValue()* in interface *Property* (section 4).
- Added description of method *getAdjacentElement()* for class *xOiLayoutGroup* (section 9.1).
- Removed description of method *swapFields()* for class *xOiTabularPlGroup* (since it is not really directly usable in *MethodCall* actions in OAP).
- Added description of method *moveFieldElement()* for class *xOiTabularPlGroup* (section 10.3).

Version 1.8 (December 6, 2021)

- Added description of method *getChID()* for class *GoMetaType* (section 6).

Version 1.7 (October 19, 2021)

- Reorganization of document structure.
- Added description of method *hasProperties()* in interface *Property* (section 4).
- Added section covering functionality common for all planning group classes (section 7).
- Added description of method *updateCommonProps()* implemented in all planning group classes (section 7.3).
- More precise description of method *replaceElement()* and added description of method *replaceElement2()* for class *xOiJointPlGroup* (section 8.2).
- Added description of methods *flipElement()*, *isFlippedElement()*, *swapNeighbors()*, *canSwapNeighbors()*, *setAdjustSide4Insertion()* and *setAdjustSide4Removal()* for class *xOiLayoutGroup* (section 9.4).
- Added section for class *xOiTabularPlGroup* (section 10).
- Added appendix listing the methods per object type (appendix B).
- Added this history appendix.