EasternGraphics
visualize your business

Application Note (2023-11-07)
Hinweise und Tipps zur OAP-Datenanlage

Inhalt

Literatur

[an0601] Application Note zu den Steuerdatentabellen (AN-2006-01). EasternGraphics GmbH

[article] Die OFML-Schnittstellen *Article* und *CompositeArticle* (Spezifikation). EasternGraphics GmbH

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

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

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

[ofml] OFML – Standardisiertes Datenbeschreibungsformat der Büromöbelindustrie.
Version 2.0, 3. überarbeitete Auflage.
Industrieverband Büro- und Arbeitswelt e.V. (IBA)

[property] Die OFML-Schnittstelle *Property* (Spezifikation). EasternGraphics GmbH

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

Bis auf [xoi] sind die Dokumente über das *pCon Download Center*

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

in der Kategorie *OFML Specifications* verfügbar.

Rechtliche Hinweise

1 Allgemeines

1.1 Tabellenerweiterungen / EBase

Im Zuge der Weiterentwicklung von OAP kommt es gelegentlich zu Erweiterungen bei bereits verwendeten Tabellen bzw. zur Definition neuer Tabellen. Dies bedingt zwingend die Veröffentlichung einer neuen Formatversion (Minor oder Major)[1]. Die jeweils verwendete Formatversion muss in der Tabelle *Version* hinterlegt werden, um eine korrekte Verarbeitung der OAP-Daten zu gewährleisten.

Mit jeder neuen Major- oder Minor-Version von OAP wird auch eine entsprechende neue EBase-Tabellenbeschreibungsdatei `oap_<major>_<minor>.inp_descr` bereitgestellt.

1.2 Separate Serie für OAP-Daten

Mittels des Schlüssels `oap_program` in der Registrierungsdatei einer OFML-Serie kann auf eine andere Serie verwiesen werden, in der sich die OAP-Daten für die Artikel der OFML-Serie befinden.

In folgenden Situationen ist die Anlage der OAP-Daten in einer separaten Serie notwendig bzw. angeraten:

- Die OAP-Daten beziehen sich auf Artikel aus mehreren OFML-Serien.
- Im OAP-Projekt kommen Planungsgruppen zum Einsatz.
  Der einfachste Weg, eine Planungsgruppe in den Katalog zu integrieren, ist die Anlage eines eigenen (Pseudo)-Artikels in den OCD-Daten[2]. (Ein eigener Artikel für die Planungsgruppe ist auf jeden Fall erforderlich, wenn an die Planungsgruppe OAP-Interaktoren gebunden werden sollen.) In den meisten Fällen ist es dann nicht sinnvoll, den Artikel für die Planungsgruppe zusammen mit den eigentlichen Artikeln in einer OCD-Datenbank anzulegen, es sei denn, es besteht eine so enge Kopplung zwischen OCD- und OAP-Daten, dass auch eine simultane Verteilung angezeigt ist.
- Die OAP-Daten ändern sich häufiger[3] oder in einem anderen Rhythmus als die OFML-Daten und die Vertriebsprozesse des Herstellers lassen eine separate Verteilung der OAP-Daten zu.
- OFML- und OAP-Daten werden (parallel) durch verschiedene Personen angelegt bzw. gepflegt[4].

1.3 Konventionen für Identifikatoren

In den meisten Tabellen werden Identifikatoren (Feld-Typ *ID*) als Zugriffsschlüssel verwendet. Solange noch keine Applikation zur OAP-Datenanlage vorliegt, müssen diese Schlüssel manuell definiert werden. Zur schnelleren Orientierung in den Tabellen[5] wird empfohlen, den Zweck eines Identifikators durch ein einheitliches Präfix zu kennzeichnen. Der „*OAP-Styleguide*“ enthält entsprechende Empfehlungen.

[1] Umgekehrt kann eine neue Minor-Version auch ausschließlich Erweiterungen enthalten, die keine Änderungen in den Tabellenstrukturen bedingen (z.B. neue Interaktor-Symbole).
[2] in der Regel ohne Preis und Merkmale, aber ggf. mit Artikel-Kurztext
[3] was zumindest für die Anfangsphase der meisten OAP-Projekte zutreffen dürfte
[4] im Fall von OAP z.B. auch durch einen externen Partner
[5] für den Datenanleger selber, aber auch für Support-Mitarbeiter

2 Bedingungen/Ausdrücke

2.1 OAP-Ausdrücke vs. OCD-Ausdrücke

Die Syntax von Ausdrücken im OAP unterscheidet sich von der Syntax in OCD-Ausdrücken [ocd]. Diejenigen, die bislang nur OCD-Daten angelegt haben, müssen diesbezüglich also etwas umlernen. (Hingegen sollten Datenanleger, die schon die OFML-Programmiersprache genutzt haben, keine größeren Schwierigkeiten habe, da sich die Syntax stark ähnelt.)

OAP-Ausdrücke sind ausführlich im Anhang A der OAP-Spezifikation beschrieben. Für den Anfang kann man sich dabei auf den Abschnitt A.4 konzentrieren. Zum schnelleren Umstieg von OCD auf OAP hier ein Überblick über die in **Bedingungen** (logischen Ausdrücken) jeweils verwendeten Operatoren:

Operator	OCD	OAP
Oder-Verknüpfung	`OR`	`||`
Und-Verknüpfung	`AND`	`&&`
Ist kleiner als	`<` (`LT`)	`<`
Ist kleiner gleich	`<=` (`LE`)	`<=`
Ist gleich	`=` (`EQ`)	`==`
Ist ungleich	`<>` (`NE`)	`!=`
Ist größer gleich	`=>` (`GE`)	`>=`
Ist größer als	`>` (`GT`)	`>`

Des Weiteren ist zu beachten, dass Ausdrücke im OCD mit OCD-Merkmalen operieren, Ausdrücke im OAP hingegen mit OFML-Properties. Das wirkt sich insbesondere auf OCD-Merkmale mit dem Datentyp `C` (Char) aus: wenn diesen eine Werteliste zugeordnet ist, wird für diese Merkmale eine OFML-Property generiert, deren Werte Symbole sind (also keine Zeichenketten). Zum Vergleich hier zwei inhaltlich identische Bedingungen in OCD und OAP:

OCD: `Legs = 'ALU'`

OAP: `Legs == @ALU`

Ein weiterer signifikanter Unterschied ist die Darstellung von Zeichenketten-Literalen (Konstanten): Im OCD werden diese in einfache Anführungszeichen eingeschlossen (s. Beispiel oben), im OAP hingegen in doppelte Anführungszeichen (`"ALU"`).

2.2 Verwendung von Standardmethoden

In Ausdrücken (zur Bestimmung der Position von Interaktor-Symbolen oder zur Formulierung von Bedingungen für Interaktoren oder Aktionen) sollte nach Möglichkeit die Verwendung von projekt-spezifisch implementierten Methoden vermieden und stattdessen auf Property-Werte und/oder Standardmethoden zurückgegriffen werden.

Im begleitenden Dokument *„Useful OFML methods for OAP data“* [methods] sind nützliche Standardmethoden beschrieben.

Der folgende Abschnitt zeigt ein Beispiel zur Verwendung von Standardmethoden.

2.3 Artikel auf oberster Planungsebene?

Bei der OAP-Datenanlage besteht oft die Anforderung, dass bestimmte Interaktoren *nicht* gültig sind (nicht angezeigt werden sollen), wenn sich der betreffende Artikel auf der obersten Planungsebene befindet, also wenn er kein Unterartikel eines *kompositen Artikels*[6] ist.

Anstatt in der (spezifischen) Klasse des Artikels eine spezielle Methode zu programmieren, die diese Bedingung prüft, kann man hierzu auch auf die Standardmethoden *getFather()* und *getRoot()* zurückgreifen[7].

oap_methodcall.csv:

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

Mit entsprechenden Aktionen (Tabelle *Action*) kann die Bedingung in der *Interactor*-Tabelle dann wie folgt formuliert werden:

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

bzw. wenn es mit einer Bedingung für den Fall verknüpft werden soll, dass der Artikel ein Unterartikel eines kompositen Artikels ist:

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

2.4 Numerische Metamerkmale

In den Metadaten mit den Formaten *chi* bzw. *chf* (Tabelle `go_types`) angelegte Merkmale zur Auswahl eines Wertes aus einer Liste von ganzen Zahlen bzw. Gleitkommazahlen liefern aus historisch-technischen Gründen über die Property-Schnittstelle die Zeichenketten-Darstellung des aktuellen Wertes, also z.B. anstatt der Zahl `1200` die Zeichenkette `"1200"`.

Diese Merkmale können somit nicht *direkt* im OAP in Ausdrücken zum Vergleich mit numerischen Werten oder zur Bestimmung einer Koordinate für die Position von Interaktor-Symbolen verwendet werden! Stattdessen müssen die Funktionen *int()* bzw. *float()* zur Umwandlung des Zeichenkettenwertes in eine ganze Zahl bzw. eine Gleitkommazahl verwendet werden.

2.5 Spezielle Symbole

Gelegentlich müssen in Ausdrücken Symbole angegeben werden, die nicht als Symbol-Literal in der Normalform dargestellt werden können. Die aus OFML bekannte Alternative der Verwendung des Symbol-Konstruktors `Symbol()` ist im OAP nicht anwendbar! Stattdessen muss die Konvertierungsfunktion `symbol()` oder die alternative Form zur Darstellung eines Symbol-Literals verwendet werden (s. Anhang A.3.2 in der OAP-Spezifikation).

Ein Beispiel anhand einer *PropChange*-Aktion:

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

oder

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

[6] z.B. einer Planungsgruppe (s. Abschn. 5)
[7] Zur Bedeutung der beiden Methoden siehe [methods] oder [ofml].

2.6 Ausdrücke mit ganzen Zahlen

Das im Anhang A.4.10 der OAP-Spezifikation beschriebene Verhalten in Bezug auf binäre arithmetische Operatoren hält eine kleine "Stolperfalle" bei Verwendung von ganzzahligen Operanden (Typ *Int*) bereit, speziell bei der Division:

> "Wenn beide Operanden einen numerischen Typ haben und mindestens ein Operand vom Typ *Float* ist, wird der andere Operand ggf. nach *Float* konvertiert und die Berechnung in *Float* durchgeführt. Das Ergebnis ist dann ebenfalls vom Typ *Float*. Andernfalls hat das Ergebnis den gleichen Typ wie beide Operanden."

Mit anderen Worten: Haben beide Operanden den Typ *Int*, hat auch das Ergebnis den Typ *Int*. Bei der Division entfällt dann also der eventuell anfallende nicht-ganzzahlige Rest der Division!

Das ist insbesondere bei der Positionierung von Interaktor-Symbolen (Tabelle *NumTripel*) bei Verwendung von Properties des Typs "i" zu beachten!

Beispiel:

Das aktive Objekt besitze eine Property TIEFE vom Typ "i", welche die aktuelle Tiefe des Objektes in Millimetern angibt. Das Interaktor-Symbol soll Tiefen-mittig platziert werden. Da die entsprechende Z-Koordinate in Metern angegeben werden muss, würde man also evtl. folgenden Ausdruck angeben:

```
TIEFE / 2000
```

Bei einer aktuellen Tiefe von z.B. 650 mm liefert der Ausdruck aber 0, anstelle der gewünschten 0.325!

Der Ausdruck muss also wie folgt formuliert werden:

```
TIEFE / 2000.0
```

Oder man formt den Ausdruck in eine Multiplikation um:

```
TIEFE * 0.0005
```

3 Interaktoren

3.1 Empfehlungen zur Verwendung der 3 abstrakten Symbolgrößen

large	Interaktoren auf Ebene eine Planungsgruppe
medium	Aktionen, die sich auf Top-Level-Objekte oder Elemente einer Planungsgruppe beziehen
small	Aktionen, die sich auf Kind-Objekte (Unterartikel) beziehen

3.2 Interaktor an Planungsgruppe oder Gruppenelementen?

Kommt im OAP-Projekt eine Planungsgruppe (s. Abschn. 5) zum Einsatz, können bestimmte Aktionen sowohl über einen Interaktor an der Gruppe als auch über Interaktoren an den Elementen ausgelöst werden.

> Ein typisches Beispiel ist das Anfügen und Entfernen von Elementen. In der Regel ist dies nur an bestimmten Stellen (Elementen) des Layouts erlaubt. Die Interaktoren mit den entsprechenden Aktionen könnten dann an den Elementen, wo die Aktion möglich ist, angezeigt werden. Es wäre aber auch möglich, dass die Gruppeninstanz selber die Interaktoren an den erlaubten Stellen (Elementen) platziert.

Bei der Entscheidung müssen sowohl Aspekte der Nutzerführung als auch die jeweiligen Aufwände berücksichtigt und gegeneinander abgewogen werden (ggf. in Absprache mit dem Auftraggeber). Nachfolgend einige Überlegungen zur Entscheidungsfindung.

Aus **Anwendersicht** sind Interaktoren an den Gruppenelementen insofern problematisch, als dass sie erst sichtbar sind, wenn der Anwender das betreffende Element selektiert hat. Dem ungeübten Anwender ist aber oft gar nicht bewusst/bekannt, dass er erst Elemente selektieren muss, um bestimmte Aktionen ausführen zu können. Im Szenario des Anfügens eines neuen Elements (s.o.) kommt hinzu, dass der Anwender oft weiter an das neue Element anfügen möchte und dieses dazu auch erst wieder selektieren muss.
Auf der anderen Seite können viele Interaktoren an der Gruppe zur Übersättigung und bei größerer Entfernung des Betrachters zu unschönen Überlagerungseffekten führen. Um die Übersättigung der Gruppe zu vermeiden bzw. zu verringern, könnte ein Kompromiss darin bestehen, dass Interaktoren, die nur bei bestimmten Elementen sichtbar/gültig wären (s. Beispiel oben), an die Gruppen gebunden werden. Interaktoren hingegen, die für (fast) alle Elemente gültig sind, werden auch direkt an diese gebunden.

Element-bezogene Interaktoren (Aktionen) an der Gruppe implizieren in der Regel einen höheren **Aufwand**, da die Positionen für die Interaktor-Symbole ausgehend von der Position (und Rotation) des betreffenden Elementes innerhalb der Gruppe ermittelt werden müssen (wozu entsprechende Methoden in der projekt-spezifischen Klasse für die Planungsgruppe implementiert werden müssen). Weiterer Aufwand entsteht, wenn die Gruppeninstanz selber nicht die erforderliche Funktionalität (Property, Methode) zur Ausführung der gewünschten Aktion besitzt. In der projekt-spezifischen Klasse müssen dann extra Methoden implementiert werden, die die entsprechende Funktionalität des betreffenden Elementes nutzen/aufrufen.
Bei Interaktoren an den Elementen selber besteht hingegen ggf. Aufwand für die Formulierung der Gültigkeitsbedingung, wenn der Interaktor nur an bestimmten Elementen sichtbar sein soll[8]. Hierbei können jedoch oft

[8] Das korrespondiert mit der Empfehlung aus Anwendersicht oben, diese Interaktoren doch eher an die Gruppe zu binden.

vorhandene Standardmethoden verwendet werden, d.h. die Formulierung dieser Bedingungen kommt u.U. ohne projekt-spezifische Programmierung aus.

Falls die Entscheidung gefallen ist, bestimmte Element-bezogene Aktionen über Interaktoren an der Gruppe auszuführen, die Anzahl der möglichen Interaktoren und die Positionen der Interaktor-Symbole aber schwer überschaubar sind, kann/sollte die Verwendung von *dynamischen Interaktoren* in Erwägung gezogen werden (siehe folgenden Abschnitt).

Tipp:
Gemäß der Empfehlung aus dem vorhergehenden Abschnitt sollten Interaktoren der Planungsgruppe, die sich auf einzelne Elemente beziehen, eine kleinere Symbolgröße haben als die Interaktoren, die sich tatsächlich auf die Gruppe als Ganzes beziehen.

3.3 Dynamische Interaktoren

Diese werden durch Implementierung der (parameterlosen) Methode ***getDynamicOAPInteractors()*** in der Klasse der betreffenden Artikelinstanz (Planungsgruppe) realisiert[9]. Die Methode wird von der Applikation bei der Selektion der Instanz und nach Abarbeitung der Aktionen eines aktivierten Interaktors gerufen[10]. Die von der Methode gelieferten Interaktoren werden dann *zusätzlich* zu den aktuell gültigen statischen Interaktoren angezeigt.

Der Rückgabewert der Methode muss vom OFML-Typ *List* sein, dessen Elemente vom Typ *Vector* sind und (jeweils) folgende 2 Elemente enthalten:

1. Interaktor ID *(String)*
2. Interaktor-Information *(Vector)*

Die ID für einen dynamischen Interaktor kann frei festgelegt werden, muss jedoch eindeutig sein und darf nicht mit der ID eines statischen Interaktors übereinstimmen!

Die Interaktor-Information im 2. Element enthält folgende Elemente:

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

Die Elemente 1-4 korrespondieren mit den entsprechenden Feldern der OAP-Tabelle *Interactor*.

Die Elemente in der SymbolDisplay-Info (Element 5) sind wieder vom Typ *Vector* und definieren je ein Symbol anhand folgender Elemente (die mit den entsprechenden Feldern der OAP-Tabelle *SymbolDisplay* korrespondieren):

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

[9] sind theoretisch also auch bei „normalen“ Artikeln möglich
[10] s.a. Methode *xOiOAPManager::getInteractors()*, Abschn. 6.1

Anmerkungen:

- Da es kein Element für eine Gültigkeitsbedingung gibt, muss/darf die Methode nur die Interaktoren liefern, die auch zum Zeitpunkt des Methodenaufrufs gültig sind.
- Die Informationen zu den Aktionen, die anhand ihrer ID spezifiziert sind, müssen in den OAP-Daten hinterlegt sein. Weitere Anmerkungen zu diesem Punkt siehe unten.
- *SymbolType* und *SymbolSize* müssen als *Symbol* angegeben werden, dessen Wert dem Bezeichner des gewünschten Symbol-Typs bzw. der gewünschten Symbol-Größenstufe gemäß OAP-Spezifikation entspricht.
- Die Elemente *NeedsPlanMode* und *HiddenMode* dürfen keine Ausdrücke enthalten, sondern müssen als expliziter boolescher Wert (`0` oder `1`) angegeben werden.
- *Float*-Werte müssen explizit angegeben werden, d.h., auch hier sind keine Ausdrücke möglich (und es kann auch nicht die ID eines Eintrags in der Tabelle *NumTripel* verwendet werden).

Wie im 2. Punkt oben angemerkt, müssen die Aktionen des dynamischen Interaktors in den OAP-Daten hinterlegt werden. Wenn eine Aktion inhaltlich (semantisch) identisch für alle relevanten Planungselemente ist, kann der Platzhalter **`$INTERACTOR`** verwendet werden. Ansonsten müsste für jedes relevante Planungselement je eine eigene Aktion mit dem passenden Target-Objekt (Objektkategorie *MethodCall*) bzw. einem passenden Objekt als Argument einer *MethodCall*-Aktion angelegt werden. Bei einer beliebigen bzw. unbekannten Anzahl von relevanten Elementen ist das praktisch unmöglich.

Die Grundidee bei der Verwendung des Platzhalters `$INTERACTOR` ist, die ID eines für ein gegebenes Element definierten dynamischen Interaktors so aufzubauen, dass daraus das relevante Element wieder herausgelesen werden kann. Das kann z.B. bewerkstelligt werden[11], indem in die ID des Interaktors der lokale Objektname des Elements innerhalb der Gruppe codiert wird. Dazu stehen die Methoden *localObjName()* und *localName2Obj()* in der Basisklasse *xOiPlGroup* für alle Typen von Planungsgruppen zur Verfügung, siehe Beispiel.

Das **Beispiel** zeigt die Realisierung von dynamischen Interaktoren in einer vom Typ *xOiJointPlGroup* abgeleiteten Planungsgruppenklasse, die es ermöglichen sollen, mittels einer *PropEdit*-Aktion die Länge jedes Layout-Elementes ändern zu können:

oap_object.csv:

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

oap_action.csv:

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

oap_methodcall.csv:

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

[11] Eine andere Möglichkeit wäre, in die ID des Interaktors den Index des Elementes in der Liste der Layout-Elemente zu codieren.

barplgroup.cls:

```
static var sIA_PE_Length_ID_Prefix = "IA_PE_Length_";

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

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

    return(tRet);
}

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

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

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

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

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

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

Hinweis:
Aktuell ist die Objektkategorie *MethodCall* nicht für das Target-Objekt einer Aktion erlaubt, deren ID als Argument eines Aufrufs der Funktion *methodCall()* verwendet wird (siehe [oap]).
Die in dem obigen Beispiel definierte Objekt-ID `OB_dynID` kann somit nicht für Target-Objekte von Aktionen verwendet werden, die als Argumente von *methodCall()*-Ausdrücken in Bedingungen zur Anwendung kommen!

Angenommen, in dem Beispiel oben soll zusätzlich zur *PropEdit*-Aktion eine *ActionChoice*-Aktion mit einer Auswahl anzufügender Objekte zur Anwendung kommen, wobei die Auswahl abhängig von dem aktuell selektierten Gruppenelement ist. Ob ein bestimmtes Objekt aus der möglichen Auswahlliste an das selektierte Gruppenelement angefügt werden kann, soll mittels der Methode *canAddObject(pType)* der Klasse der Gruppenelemente bestimmt werden.

Die Bedingung für eine Option in der Tabelle `ActionList` würde dann wie folgt formuliert:

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

Da für das Target-Objekt dieser Aktionen nicht die Objektkategorie *MethodCall* verwendet werden kann, darf der Methodenaufruf nicht direkt auf dem Gruppenelement erfolgen. Stattdessen muss eine (gleichnamige) Methode der Klasse der Planungsgruppe gerufen werden, die ihrerseits dann die Methode auf dem relevante Gruppenelement aufruft. Dabei kommt der Platzhalter `$INTERACTOR` und die schon oben beschriebene Methode `getElemByDynIID()` zum Einsatz:

oap_action.csv:

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

oap_methodcall.csv:

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

barplgroup.cls:

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

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

    return tRet;
}
```

3.4 3D-Interaktor-Symbole

Bei Interaktor-Symbolen, die mittels Pfeildarstellung die Wirkungsrichtung der mit dem Interaktor verbundenen Aktion veranschaulichen, ist die Anwendung als 3D-Symbol in Betracht zu ziehen.

Bsp.:
An der rechten Seite eines Objektes befindet sich ein Interaktor mit Symboltyp *ChangeDim2Right*, an den eine Aktion gebunden ist, welche die Breite des Objektes vergrößert.

Ein „normales“ 2D-Symbol, welches immer parallel zur Bildschirmebene liegt, würde immer nach rechts zeigen. Das würde dem Anwender (bei einem nicht rotierten Objekt) nur in der Vorderansicht bzw. Draufsicht die korrekte Wirkrichtung veranschaulichen.

Da bei 3D-Symbolen die Ebene festgelegt werden muss, in der das (flache) Symbol-Icon liegen soll, muss man sich für eine Ansicht entscheiden, die zur Bearbeitung des Objektes am geeignetsten ist:

- Bei Objekten mit einer geringen Höhe ist in der Regel die Draufsicht die geeignetste, d.h., das Symbol-Icon muss dann in der X-Z-Ebene des Objekts liegen.
- Bei Objekten mit einer geringen Tiefe ist in der Regel die Vorder- bzw. Rückansicht die geeignetste, d.h., das Symbol-Icon muss dann in der X-Y-Ebene des Objekts liegen.

Ein für die Draufsicht vorgesehenes 3D-Symbol würde folgendermaßen definiert werden[12]:

oap_numtripel.csv:

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

oap_symboldisplay.csv

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

Erklärung:
Der Richtungsvektor des Sichtbarkeitsbereiches (`NT_POS_Y_AXIS`, Feld 5) verläuft entlang der positiven Y-Achse (Draufsicht). Das 3D-Symbol liegt somit in der X-Z-Ebene des Objekts und ist aufgrund des Eintrags im Feld 7 immer entlang der positiven X-Achse des lokalen Koordinatensystems des Objektes ausgerichtet.

Anmerkungen zum Öffnungswinkel im Feld 6:

Ohne Angabe eines Öffnungswinkels (leeres Feld 6) ist kein Sichtbarkeitsbereich definiert, womit auch der möglicherweise vorhandene Eintrag im Feld 5 für den Richtungsvektor nicht wirksam wird. Das Ergebnis wäre dann ein normales 2D-Symbol! Soll der Sichtbarkeitsbereich tatsächlich nicht eingeschränkt werden, muss also, wie in dem Beispiel oben, ein Öffnungswinkel von 360 Grad angegeben werden. Da das 3D-Symbol immer „flacher“ und damit unerkenntlicher wird, umso mehr sich der Betrachtungswinkel der X-Z-Ebene des Objektes nähert, ist jedoch für 3D-Symbole eine Einschränkung der Sichtbarkeitsbereiches empfehlenswert, z.B.:

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

Der Interaktor hätte dann 2 Symbole mit sich gegenseitig ausschließenden Sichtbarkeitsbereichen, welche bei einem Betrachtungswinkel von weniger als 15 Grad zur X-Z-Ebene des Objektes nicht sichtbar wären. (Der zweite Eintrag kann entfallen, wenn der Interaktor nur sichtbar sein soll, wenn der Anwender von oben auf das Objekt schaut.)

Ein für die Vorder- und Rückansicht vorgesehenes 3D-Symbol (für das Beispiel mit Symboltyp *ChangeDim2Right*) würde entsprechend wie folgt definiert werden:

oap_numtripel.csv:

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

oap_symboldisplay.csv

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

[12] s.a. Abb. 2 in Abschn. 4.6 der OAP-Spezifikation

3.5 Korrekter Konfigurationskontext?

Die Gültigkeitsbedingung in der OAP-Tabelle *Interactor* kann bzw. muss dazu genutzt werden, damit der Interaktor auch nur dann angezeigt wird, wenn es der aktuelle Konfigurationskontext zulässt.

Eine häufige Anforderung ist, dass ein Interaktor nur angezeigt werden soll, wenn der betreffende Artikel ein Unterartikel eines *kompositen Artikels* ist, also wenn er sich *nicht* auf der obersten Planungsebene befindet. Das trifft z.B. auf folgende Fälle zu:

- Ein *Add*-Interaktor zum Anfügen einen Nachbar-Elementes im Rahmen einer Planungsgruppe.
  (s.a. Abschn. 3.2)
- Alle *Delete*-Interaktoren!
  (s.a. Abschn. 4.4)

Eine typische bzw. empfohlene Lösung für diese Anforderung ist in Abschn. 2.3 beschrieben.

Des Weiteren muss beachtet werden, dass manche Applikationen die Funktion „***Artikel aufspalten***“[13] besitzen, bei deren Ausführung ein kompositer Artikel in seine Bestandteile, d.h. individuelle Artikel auf oberster Planungsebene, zerlegt wird.

Handelt es sich bei dem kompositen Artikel um den Spezialfall einer **Metatyp**-Instanz, ist diese nach der Ausführung der Funktion *„Artikel aufspalten“* nicht mehr existent!
Eine Konsequenz davon ist, dass nach dem Aufspalten nicht mehr auf Metamerkmale zugegriffen werden kann. Interaktoren mit *PropEdit*- oder *PropChange*-Aktionen, die Metamerkmale verwenden, dürfen dann entsprechend nicht mehr angezeigt werden! Besitzt ein OAP-Typ ausschließlich derartige Interaktoren, kann dies auf einfache Weise erreicht werden, indem der OAP-Typ (ausschließlich) in der Mapping-Tabelle *Metatype2Type* einem (oder mehreren) Metatypen (ID) zugeordnet wird[14].

[13] o.ä., z.B. *„Artikel aufbrechen“*
[14] d.h., es sollte/darf keine Zuordnung zu Artikeln stattfinden (Mapping-Tabelle *Article2Type*)

4 Aktionen

4.1 Artikelerzeugung mittels *xOiCreateArticle()*

In den Fällen, wo zur Artikelerzeugung keine Aktion vom Typ *CreateObj* verwendet werden kann, und die Artikelerzeugung also programmiert werden muss, empfiehlt sich die Verwendung der globalen Funktionen *xOiCreateArticle()* bzw. (ab den Herbst-Releases 2023) *xOiCreateArticle2()*[15].

Im Default-Modus *@extended* simulieren die Funktionen den Standardprozess der Artikelerzeugung in den Applikationen von EasternGraphics, wobei zur Ermittlung von Position und Rotation der neuen Artikelinstanz der sogenannte *checkAdd()*-Mechanismus verwendet wird (inkl. Erzeugung einer temporären Artikelinstanz).

Wenn Position und Rotation der neuen Artikelinstanz bei der programmierten Artikelerzeugung selber festgelegt werden, sollten die Funktionen im Modus *@lite* verwendet werden, da dieser eine bessere Performanz bietet. Position und Rotation müssen dann nach Aufruf der Funktion der von der Funktion zurückgegebenen Artikelinstanz zugewiesen werden.

Wie Aktionen vom Typ *CreateObj* erfordern die Funktionen *xOiCreateArticle()* und *xOiCreateArticle2()* die Angabe eines übergeordneten Artikels (Vaterobjekt) und sind somit nicht zum Einfügen auf der obersten Planungsebene geeignet.

4.2 Änderungsrichtung bei *DimChange*

Ab OAP 1.2 kann im Feld *Dimension* der Tabelle *DimChange* festgelegt werden, in welcher Achsenrichtung eine Änderung erlaubt ist. Zuvor war nur eine Änderung in positiver Richtung der jeweiligen Dimensionsachse möglich (implementiert). Zur Gewährleistung der Abwärtskompatibilität werden bei Daten, die gemäß einer älteren OAP-Version angelegt sind, die Werte `X`, `Y` und `Z` entsprechend wie `PX`, `PY` bzw. `PZ` gemäß OAP 1.2 behandelt.

Bei den Dimensionen X und Z ist es in den meisten Fällen sinnvoll, eine Änderung in beiden Richtungen zu erlauben (d.h., ab OAP 1.2 die Werte `X` bzw. `Z` anzugeben), um dem Anwender unabhängig von Rotation des Objektes bzw. Kamera-Perspektive ein optimales Veränderungsverhalten anzubieten. Gegebenenfalls sollten dazu existierende Daten nach OAP 1.2 portiert werden[16].

4.3 Verschachtelung von Methodenaufrufen

Da die Argumente für einen Methodenaufruf (Tabelle *MethodCall*) als Ausdrücke angegeben werden können, können mittels des *methodCall()*-Ausdrucks auch verschachtelte Methodenaufrufe realisiert werden.

Beispiel:

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

[15] In der XOI-Dokumentation [xoi] findet man diese Funktionen in der Sektion „xOiFuncs“ im Abschn. „Miscellaneous utilities“.
[16] wobei die neuen Felder *Separate* und *ThirdDim* in der Tabelle *DimChange* zu beachten wären

4.4 Hinweise zu *DeleteObj*

Von den aktuell unterstützten Objektkategorien können für die Target-Objekte einer *DeleteObj*-Aktionen nur *Self* und *MethodCall* verwendet werden[17].

MethodCall ist sinnvoll im Rahmen von Interaktoren, die an Planungsgruppen (s. Abschn. 5) gebunden sind. Der Methodenaufruf kann dann dazu genutzt werden, das Element der Planungsgruppe zu bestimmen, welches entfernt werden soll.

Bei Verwendung von ***Self*** dürfen nach der *DeleteObj*-Aktion keine weiteren Aktionen folgen, die wieder ein Ziel-Objekt oder eine Objektdefinition als Parameter benötigen, da nach der Aktion das ursprüngliche aktive Objekt nicht mehr existiert!
(Das gilt nicht nur für *DeleteObj*-Aktionen, sondern generell für alle Aktionen, die zum Entfernen des aktiven Objekts führen.)

Das ist problematisch, wenn nach dem Entfernen des Objektes noch Aktionen ausgeführt werden müssten, die bestimmte Prüfungen und ggf. Nachbehandlungen durchführen[18]. Eine derartige Aufgabenstellung besteht typischerweise im Rahmen von Planungsgruppen. Wenn die *DeleteObj*-Aktion nicht doch im Rahmen eines Interaktors ausgeführt werden kann, der an die Planungsgruppe gebunden ist[19], bleibt nichts anderes übrig, als die *DeleteObj*-Aktion durch eine *MethodCall*-Aktion für die Instanz der Planungsgruppe (Objektkategorie *ParentArticle*) zu ersetzen, welche das Löschen des Elementes *und* die nachträglichen Prüfungen und Behandlungen durchführt. Das zu löschende Element wird dabei in einem Parameter mittels des Platzhalters `$SELF` übergeben:

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

Anmerkung:
Falls sich die durchzuführenden Prüfungen auf die Nachbarschaftsbeziehungen von Layout-Elementen der Planungsgruppe beziehen, könnte ein OAP-Datenanleger geneigt sein, vor der *DeleteObj*-Aktion bestimmte Methoden zur Manipulation der internen Layout-Struktur der Instanz der Planungsgruppe zu rufen, um gewünschte Ergebnisse bei den Nachbarschaftstests zu erhalten (Ausschluss des zu löschenden Elementes). Das ist jedoch fehleranfällig, da es genaue Kenntnis der Abläufe in den Basisklassen voraussetzt. Da zudem eh schon eine Methode zur Durchführung der Prüfungen in der (abgeleiteten) Klasse der Instanz der Planungsgruppe angelegt werden muss, kann/sollte diese besser auch gleich wie oben beschrieben erweitert werden.

[17] *ParentArticle* und *TopArticle* scheiden aus, da das aktive Objekt keine übergeordnete Artikelinstanz löschen kann/darf.
[18] und sich diese Aktionen auch nicht vor die *DeleteObj*-Aktion verlagern lassen
[19] und dann die Objektkategorie *MethodCall* zur Bestimmung des Target-Objektes verwendet wird

4.5 RG-Merkmale

OFML-Properties, die für OCD-Merkmale mit Geltungsbereich **RG** generiert werden, können nur dann mittels Aktionen der Typen *PropChange* und *PropEdit* geändert werden, wenn in der Steuerdatentabelle `epdfproductdb` für die Option *@NeedValuesForRGProps* der Wert `1` angegeben ist[20].

Achtung:
Eine via *@NeedValuesForRGProps* ermöglichte Zuweisung eines Wertes an ein RG-Merkmal darf (in Folge von Abhängigkeiten im OCD-Produktbeziehungswissen) nicht zur Änderung eines Wertes eines konfigurierbaren Merkmals führen, das in der Merkmalsliste nach dem RG-Merkmal folgt! Ansonsten kann es zu Problemen („Einfrieren“) bei der (Wieder)Erzeugung der Artikelinstanz kommen (z.B. im Rahmen einer Artikelaktualisierung)[21].

4.6 Aktionstyp *NoAction*

Ein Interaktor wird nicht angezeigt, wenn er aktuell keine gültige Aktion hat. Dies kann in einer frühen Phase der Projektentwicklung ein Handicap sein, wenn zunächst nur die Interaktor-Symbole erstellt und positioniert werden sollen. In diesem Fall kann der spezielle Dummy-Aktionstyp *NoAction* verwendet werden, bei dem keine weiteren Daten angegeben werden müssen.

Beispiel:

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

[20] Durch den Wert `1` enthält die Werteliste der generierten Property alle Werte, die für das RG-Merkmal in der OCD-Wertetabelle hinterlegt sind. Ohne die Option (bzw. beim Wert `0`) enthält die Werteliste nur den aktuellen Wert, womit die Property praktisch read-only ist.
[21] da RG-Merkmale nicht im Variantencode gespeichert werden

5 Planungsgruppen

Die Ausführungen in diesem Abschnitt beziehen sich auf die in OAP-Projekten häufig verwendeten Klassen zur Realisierung von Planungsgruppen aus der OFML-Basis-Bibliothek XOI[22].

Nützliche, direkt in OAP via *MethodCall*-Aktion verwendbare Methoden dieser Klassen sind im Dokument [methods][23] beschrieben. Dieses Dokument enthält auch jeweils eine kurze Einführung in den Zweck und die Anwendung der Planungsklassen.

Mit den Steuerdatentabellen `customplgroup`, `jointplgroup`, `layoutgroup` bzw. `tabularplgroup` können einige Aspekte des Verhaltens von Planungsgruppen gesteuert werden. Die möglichen Optionen sind in der Application Note AN-2006-01 beschrieben [an0601].

Hinweis zur Verwendung von *xOiJointPlGroup* vs. *xOiLayoutGroup*:
Eine reine links-rechts-orientierte Anschlussplanung kann prinzipiell auch mittels *xOiLayoutGroup* realisiert werden[24]. Die Verwendung von *xOiJointPlGroup* ist für derartige Planungsgruppen in den meisten Fällen jedoch die „leichtgewichtigere“ Lösung und somit vorzuziehen. Bestimmte Anforderungen, die über eine reine links-rechts-orientierte Anschlussplanung hinausgehen, lassen sich in Grenzen auch noch mit *xOiJointPlGroup* umsetzen, siehe Abschnitte 5.6 und 5.7.

5.1 Verwendung der Objektkategorie *MethodCall*

Insbesondere im Zusammenhang mit Planungsgruppen ergeben sich für die Objektkategorie *MethodCall* einige sinnvolle Einsatzmöglichkeiten. Ein häufiges Anwendungsszenario ist z.B., dass auf einem via *CreateObj*-Aktion erzeugten Element der Planungsgruppe mittels einer *PropChange*-Aktion noch eine bestimmte Property gesetzt werden muss/soll. Das Target-Objekt für diese *PropChange*-Aktion kann in dem Fall nicht mittels der Objektkategorie *Self* bestimmt werden (da sich diese ja auf das Objekt bezieht, an das der Interaktor gebunden ist, dessen Aktionen gerade abgearbeitet werden). Wenn an das letzte Layout-Element der Planungsgruppe angefügt wurde, kann man sich aber im Fall der Klasse *xOiJointPlGroup* z.B. der Methode *lastObj()* bedienen. In den Klassen *xOiLayoutGroup* und *xOiCustomPlGroup* gibt es äquivalente Methoden[25].

Für das obige Beispiel könnte die *PropChange*-Aktion dann etwa wie folgt realisiert werden:

oap_object.csv:

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

oap_action.csv:

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

oap_methodcall.csv:

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

[22] Aktuell sind das konkret die Klassen *xOiCustomPlGroup*, *xOiJointPlGroup*, *xOiLayoutGroup* und *xOiTabularPlGroup*. Diese sind in der XOI-Dokumentation [xoi] in der Sektion „Planning Groups“ spezifiziert.
[23] nur in Englisch verfügbar
[24] Die Gruppe würde in dem Fall nur aus einem Branch bestehen.
[25] siehe Dokument [methods]

Hinweis:
Eine Methode, die in der Objektkategorie *MethodCall* verwendet wird, muss eine Instanz liefern, die einen Artikel repräsentiert[26]. Für spezielle Hilfsobjekte innerhalb von Planungsgruppen ist diese Voraussetzung unter Umständen nicht erfüllt. Die Lösung besteht dann darin, die Aktion nicht direkt auf diesem Hilfsobjekt auszuführen, sondern mittels Aufruf einer Methode auf der Gruppeninstanz (*MethodCall*-Aktion), wobei das Hilfsobjekt mittels *methodCall()*-Ausdruck als Argument an diese Methode übergeben wird[27] (s.a. Abschn. 4.3).

5.2 Dimensionsänderung von Gruppenelementen

5.2.1 Allgemeines

Die Klassen *xOiCustomPlGroup*, *xOiJointPlGroup* und *xOiLayoutGroup* unterstützen die Änderung der Dimension (Breite/Tiefe) von topologischen[28] Elementen der Planungsgruppe: (rechts[29]) anschließende Elemente werden "weggeschoben" bzw. "herangerückt".

Dazu muss zunächst der sogenannte *InsertMode*[30] auf `1` (oder größer) gesetzt werden[31].

Dies geschieht durch einen Eintrag

```
@InsertMode;;1
```

in der jeweiligen Steuerdaten-Tabelle.

Des Weiteren muss in den Artikel-Klassen der Gruppenelemente die Behandlung einer Dimensionsänderung durch die Planungsgruppe mittels Aufruf der Methode *dimensionChanged()* auf der Gruppeninstanz angestoßen werden[32].

Typischerweise erfolgt eine Dimensionsänderung bei Änderung des Wertes für bestimmte Properties. Entsprechend müssen diese Properties in der Methode *propsChanged()* [property] gesondert behandelt werden.

Im Fall einer von *GoMetaType* abgeleiteten Artikel-Klasse, unter der Annahme, dass eine Änderung der Property *@WidthProp* zu einer Breitenänderung führt, würde eine prototypische Implementierung dieser Methode unter Verwendung der Klasse *xOiJointPlGroup* wie folgt aussehen:

[26] Technischer Hintergrund für diese Einschränkung ist, dass die Clienten von Online-Apps manche Aktionen selber ausführen (also nicht an den Server delegieren), ihrerseits aber nur Artikel „kennen“ (keine OFML-Instanzen).
[27] Dabei kommt dieselbe *MethodCall*-Aktion zum Einsatz, die auch für die Objektkategorie *MethodCall* verwendet würde.
[28] Elemente, die das Layout der Gruppe bestimmen
[29] Im Fall von *xOiLayoutGroup* und *xOiCustomPlGroup* wird die anzupassende Seite des geänderten Elements durch einen Parameter der Methode *dimensionChanged()* festgelegt.
[30] definiert in der Basisklasse *xOiPlGroup*
[31] Bei *xOiCustomPlGroup* setzt das den (Default-)Wert `1` für die Option *@StoreNeighborship* voraus.
[32] Zu den Details (Parametern) der Methode siehe [xoi]. Im Fall von *xOiJointPlGroup* ist die Methode *dimensionChanged()* bereits in der Basisklasse *xOiLRPlGroup* implementiert und dokumentiert.

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

    if (!tRet) return (tRet);

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

    var tFather = self.getFather();

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

    return(tRet);
}
```

Für Metamerkmale, die ein natives Merkmal der gekapselten Artikelinstanz auf die Ebene der Metatyp-Instanz hochziehen (sogenannte *na-Merkmale*), funktioniert dies nicht, da eine Änderung eines solchen Metamerkmals von der Metatyp-Instanz an die gekapselte Artikelinstanz delegiert wird. In dem Fall müsste also die Methode *propsChanged()* entsprechend in der Klasse der gekapselten Artikelinstanz überschrieben werden. Da in der Standard-OFML-Datenanlage normalerweise der Standardtyp *OiOdbPlElement* verwendet wird, müsste davon also extra eine spezifische Klasse abgeleitet werden und in den OAM-Daten den entsprechenden Artikeln zugeordnet werden. Der Abschnitt 5.3 beschreibt einen anderen Lösungsansatz, mittels dessen durch eine erweiterte Metadatenanlage das Überschreiben von *propsChanged()* in der Klasse der gekapselten Artikelinstanz vermieden werden kann.

5.2.2 Beachtung von Änderungen des Begrenzungsvolumens

Der Algorithmus für die Behandlung einer Dimensionsänderung basiert – unter anderem – auf einem Vergleich von aktuellem und gespeichertem minimalen achsenorthogonalen Begrenzungsvolumen des geänderten Elements bezogen auf dessen lokales Koordinatensystem[33]. Die Speicherung des Begrenzungsvolumens erfolgt durch die Gruppeninstanz nach dem Einfüge-Vorgang eines Elements und bei jedem Aufruf der oben genannten Methoden. Findet eine Änderung des Begrenzungsvolumens statt, ohne dass eine der oben genannten Methoden gerufen wird, kommt es zu Fehlbehandlungen bei nachfolgenden Aufrufen der Methoden, da die Gruppeninstanz von der zwischenzeitlichen Änderung des Begrenzungsvolumens nichts mitbekommt hat!

Das muss in Situationen beachtet werden, wo sich zwar das Begrenzungsvolumen ändert, die eigentliche Dimension des Elementes aber unverändert bleibt und somit eigentlich keine Nachbarn verschoben werden müssen. Ein Beispiel wäre die Erzeugung eines Gestell-Fußes als Kind eines Tisches, wobei der Fuß mittig unter dem linken Nachbartisch und dem geänderten Tisch platziert wird. Zur Vermeidung des oben genannten Problems sollten die Methoden zur Behandlung einer Dimensionsänderung dann trotzdem gerufen werden, auch wenn dies einen kleinen Overhead bedeutet. Findet eine derartige Änderung ausschließlich im Rahmen des Einfügens eines Elements in die Planungsgruppe statt, kann auf den Aufruf verzichtet werden. (In dem oben genannten Beispiel etwa, wenn die Erzeugung des Fußes nur im Rahmen der Abarbeitung der Liste der Aktionen eines *Add*-Interaktors durch eine *PropChange*-Aktion erfolgt, die nach einer *CreateObj*-Aktion abgearbeitet wird, welche

[33] laut Methode *getLocalGeoBounds()* der OFML-Schnittstelle *Base* [ofml]

die eigentliche Erzeugung des Elements vornimmt.) Die Gruppeninstanz muss dann aber durch andere Methodenaufrufe unterstützt werden, wobei für die beiden o.g. Klassen jeweils verschiedene Methoden verwendet werden müssen:

- *xOiJointPlGroup*:
  Aufruf von *elementCreated()*[34] nach der relevanten *PropChange*-Aktion
- *xOiCustomPlGroup* und *xOiLayoutGroup*:
  Aufruf von *delayHandleNewElement()* vor der *CreateObj*-Aktion und von *handleNewElement2()* nach der relevanten *PropChange*-Aktion

Zu den Details dieser Methoden siehe XOI-Dokumentation [xoi].

Während die Methoden *elementCreated()* und *delayHandleNewElement()* eine einfache Signatur aufweisen und somit ohne spezielle Programmierung direkt mittels einer *MethodCall*-Aktion eingebunden werden können[35], empfiehlt sich im Fall von *handleNewElement2()* wohl eher die Kapselung durch eine speziell programmierte Methode.

5.2.3 Hinweise zum Algorithmus in *xOiJointPlGroup*

Die Methode *dimensionChanged()* führt in dem Fall, wo sich (auch) der Ursprung des lokalen Koordinatensystems geändert hat, vor der (eventuellen) Verschiebung relevanter Nachbarelemente ggf. auch eine Re-Positionierung des geänderten Elements durch. Ein Beispiel wäre das Anfügen einer Anbauplatte an der linken Seite eines Tisches. Eine Re-Positionierung des geänderten Elements findet auch statt, wenn sich die Koordinaten des relevanten Anfügepunktes entsprechend ändern.

In dem Beispiel mit der linken Anbauplatte wird also vorausgesetzt, dass der Anfügepunkt an der linken Seite des geänderten Tisches entsprechend der Breite der Anbauplatte nach links "wandert".

Ein anderes Szenario wäre das bereits oben (in Abschn. 5.2.2) erwähnte Beispiel der Erzeugung eines Gestell-Fußes mittig unter zwei benachbarten Tischen. Wird in diesem Fall *dimensionChanged()* gerufen[36], dürfen sich die Koordinaten des Anfügepunktes an der linken Seite des Tisches, an den der Fuß angefügt wird, nicht ändern. Ansonsten würde es zu einer unerwünschten Verschiebung des geänderten Tisches kommen!

Die Re-Positionierung des geänderten Elementes findet im Idealfall auf der Basis des Anfügepunktes statt, welcher beim Anfügen des geänderten Elementes (rechts) an seinen linken Nachbarn verwendet wurde. In dem Fall hingegen, wo der linke Nachbar des geänderten Elementes (links) an dieses angefügt wurde, ist dieser Anfügepunkt (mit der Standard-Implementierung in *xOiJointPlGroup*/*xOiLRPlGroup*) nicht bekannt. Als Fallback findet dann eine Verschiebung des geänderten Elementes um den Betrag der Änderung des lokalen Ursprungs statt.

Insbesondere bei abgewinkelten bzw. gebogenen Elementen kann letzteres zu einer ungewünschten Re-Positionierung des geänderten Elementes führen, wenn *dimensionChanged()* infolge einer Änderung der Ausrichtung des Elements gerufen wird (um die Re-Positionierung der Nachbarelemente zu bewerkstelligen). Das kann verhindert werden, indem mittels Überschreiben der Methode *getUsedAttPt()*[37] der relevante Anfügepunkt geliefert wird. Im Idealfall, wo die Elemente der Planungsgruppe mit Hilfe je eines Anfügepunktes an der linken und rechten Seite aneinander geplant werden und diese Anfügepunkte für alle Elemente gleich benannt sind, würde die überschriebene Implementierung etwa so aussehen:

---

[34] geerbt von der Basisklasse *xOiLRPlGroup*
[35] Im Fall von *elementCreated()* wird dabei das Argument selber auch durch Aufruf einer Methode (*lastObj()*) via Funktion *methodCall()* ermittelt, s.a. Abschnitt 4.3.
[36] vgl. auch die in Abschn. 5.2.2 beschriebene Alternative
[37] Die Methode ist in der Basisklasse *xOiLRPlGroup* definiert und dokumentiert.

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

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

    return(tRet);
}
```

5.3 Dimensionsänderung bei *na*-Metamerkmalen

Dieser Abschnitt beschreibt einen Lösungsansatz, der es mittels einer erweiterten Metadatenanlage ermöglicht, bei einer Dimensionsänderung, die auf einer Änderung eines *na*-Metamerkmals beruht, das im Abschnitt 5.2.1 beschriebene Überschreiben von *propsChanged()* in der Klasse der gekapselten Artikelinstanz zu vermeiden.

Angenommen, es gibt ein na-Merkmal `GWidth`, das die Breite des Artikels in Zentimetern angibt:

1.
In der Tabelle *go_types* wird ein ***nicht-sichtbares***, Kind-steuerndes Hilfsmerkmal (Modus 24) vom Typ (Format) „ch“ angelegt:

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

2.
In der Implementierung von *propsChanged()* in der Metatyp-Klasse (s. Abschnitt 5.2) wird der Property-Schlüssel `@GWidth2` (anstelle von `@GWidth`) verwendet.

3.
In der Tabelle *go_childprops* werden in einem entsprechenden Parametersatz die möglichen Werte für das Hilfsmerkmal hinterlegt.

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

(Der Parametersatz, hier `CHP_SE_GWidth2`, wurde zuvor in der Tabelle *go_articles* den Artikeln des Metatyps `MT_SE_Name` zugewiesen. Alternativ kann ein bereits zugewiesener Parametersatz verwendet werden.)

Tipp:
Es reicht, auch nur einen Wert anzugeben (z.B. den Startwert). Das ist besonders dann hilfreich, wenn es sehr viele mögliche Werte gibt.

4.
In der Datei `go_context.ofml` wird eine Funktion implementiert, die einen Wert des Merkmals `@GWidth` in einen Wert des Merkmals `@GWidth2` konvertiert:

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

5.
In der Tabelle *go_actions* wird eine entsprechende Aktion angelegt:

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

5.4 Gruppenspezifische Einträge in der Steuerdatentabelle

Falls in einer Serie mehrere, von einer XOI-Basisklasse abgeleitete Gruppenklassen existieren und für diese verschiedene Angaben in der jeweiligen Steuerdatentabelle gemacht werden sollen/müssen, kann zur Differenzierung das zweite Feld (Argument) der Tabelleneinträge verwendet werden, vorausgesetzt, zu den verschiedenen Gruppentypen sind auch (unterschiedliche) Artikel angelegt.

Bsp.:

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

Details siehe Application Note AN-2006-01.

5.5 Artikeltausch mittels *replaceElement()*

Edit-Interaktoren werden oft zum Artikeltausch verwendet. Idealerweise wird dabei mittels einer Aktion vom Typ *PropEdit* direkt eine entsprechende Property angesteuert. Manchmal steht aber keine geeignete Property zur Verfügung. In dem Fall kann man sich mit der in den Klassen *xOiJointPlGroup* und *xOiLayoutGroup* implementierten Methode *replaceElement()* behelfen (bzw. *replaceField()* bei einer *xOiTabularPlGroup*), welche mittels einer *MethodCall*-Aktion aufgerufen wird.

Details zur Verwendung dieser Methoden siehe Dokument [methods].

Hinweise:

- Ist der Interaktor an ein Gruppenelement gebunden (nicht an die Gruppeninstanz selber), dürfen nach der *MethodCall*-Aktion mit dem Aufruf von *replaceElement()* keine weiteren Aktionen folgen, die wieder ein Ziel-Objekt oder eine Objektdefinition als Parameter benötigen, da nach der Aktion das ursprüngliche aktive Objekt nicht mehr existiert! (Siehe auch Hinweise zu *DeleteObj* in Abschnitt 4.4.)
- Die Implementierung von *xOiLayoutGroup:: replaceElement()* funktioniert aktuell nicht, wenn ein Nachbar des zu ersetzenden Elementes zu einem anderen Branch gehört.

5.6 Nicht-Layout-Artikel bei *xOiJointPlGroup*

Gelegentlich müssen in eine Instanz von *xOiJointPlGroup* (oder einer davon abgeleiteten Klasse) Elemente eingefügt werden, die nicht Bestandteil der topologischen Liste sind, z.B. Unterbauten, Anstelltische etc.

In Bezug auf derartige Elemente einige Hinweise:

- Für Elemente der Planungsgruppe, die nicht Bestandteil der topologischen Liste sein sollen[38], muss die Hakenmethode *isValidForLRPlanning()* den Wert `0` liefern[39]. Diese Methode muss entsprechend projekt-spezifisch überschrieben werden.
  Dabei ist zu beachten, dass diese Methode unmittelbar nach Element-Erzeugung aufgerufen wird, also zu einem Zeitpunkt, zu dem der Artikel noch nicht initialisiert ist. Das bedeutet u.a., dass in der Methode nicht auf die Artikelnummer oder Properties der Instanz zurückgegriffen werden kann, um zu entscheiden, ob das übergebene Element Teil der topologischen Liste sein soll oder nicht. Es bleibt also nur die Möglichkeit, mittels *isA()* auf einen Typ zu testen oder mittels *isCat()* auf eine Kategorie[40].
- Sollen diese Elemente als Unterartikel behandelt werden, muss in der Steuerdatentabelle `jointplgroup` die Option *@AllElements4SubArticle* auf den Wert `1` gesetzt werden.
- Werden diese Elemente mittels Anfügepunkten platziert, kann in Gültigkeitsbedingungen von Interaktoren durch einen Aufruf der Methode *xOiJointPlGroup::isBusyAttPt()* festgestellt werden, ob ein Element aus der topologischen Liste ein Nachbar-Element an dem betreffenden Anfügepunkt hat[41].
  (Details zu Verwendung dieser Methode siehe Dokument [methods].)
- Bei Dimensionsänderungen von Elementen aus der topologischen Liste (s. Abschn. 5.2) werden Elemente, die nicht in dieser Liste enthalten sind, nicht verschoben! Diese müssen also ggf. neu erzeugt werden oder für die Planungsgruppe muss doch die Klasse *xOiLayoutGroup* verwendet werden.

5.7 Zusätzliche Start-Elemente

Typischerweise werden die Optionen *@StartElement* bzw. *@StartLayout* in den Steuerdatentabellen verwendet, um initial bei der Erzeugung einer Planungsgruppe bereits ein oder mehrere Start-(Layout)-Elemente zu erzeugen. Gelegentlich kann es erforderlich sein, initial auch nicht-Layout-Elemente zu erzeugen.

Da diese Initialisierung bei der Zuweisung einer Artikelnummer an die Gruppeninstanz während der Methode *setArticleSpec()* stattfindet, muss diese Methode entsprechend in abgeleiteten Klassen überschrieben werden, um zusätzliche Start-Elemente zu erzeugen. In der überschriebenen Methode muss zuerst die geerbte Implementierung aufgerufen werden. Danach können mittels der globalen Funktion *xOiCreateArticle()* (s. Abschn. 4.1) weitere Elemente erzeugt werden. Über die Methoden zum Zugriff auf die topologische Liste (*xOiJointPlGroup*[42]) bzw. zum Zugriff auf die Layout-Struktur (*xOiLayoutGroup* und *xOiCustomPlGroup*) können bei Bedarf die Referenzobjekte für die zusätzlich zu erzeugenden Start-Elemente ermittelt werden.

Beachte auch die Hinweise in Abschn. 5.10.3 sowie im Fall von *xOiJointPlGroup* die Hinweise im vorherigen Abschnitt zur Erzeugung von Elementen, die nicht Bestandteil der topologischen Liste sein sollen!

[38] Entspricht in etwa der Unterscheidung zwischen Layout-Elementen und anderen Elementen bei der *xOiLayoutGroup*.
[39] Die Methode ist in der Basisklasse *xOiLRPlGroup* definiert.
[40] Sollte dies aus irgendeinem Grund nicht möglich/ausreichend sein, muss für die Planungsgruppe die Klasse *xOiLayoutGroup* verwendet werden.
[41] Die Methoden zum Ermitteln von Nachbar-Elementen in der topologischen Liste greifen an der Stelle nicht.
[42] via Basisklasse *xOiLRPlGroup*

5.8 Löschbarkeit von Gruppenelementen

Da das Entfernen eines Elementes aus einer Planungsgruppe meist mit weiteren Aktionen einhergeht (s.a. Abschn. 4.4), sollen sie in der Regel nur per OAP-Interaktor (Symboltyp *Delete*) entfernt werden können, nicht aber via Delete-Befehl der Applikation. Früher musste dies explizit mittels direktem oder indirektem Aufruf der Methode *setCutable()* (OFML-Schnittstelle *Base* [ofml]) mit dem Wert `-1` bewerkstelligt werden. Dies ist nicht mehr erforderlich: Der gewünschte Zustand kann, getrennt für Layout-Elemente und sonstige Elemente, für alle Gruppenklassen in der jeweiligen Steuerdatentabelle mittels der Optionen *@CutableState4Layout* und *@CutableState4Other* festgelegt werden.

5.9 Gemeinsame Eigenschaften / Gruppenmerkmale

Bei allen Gruppentypen können mittels der Option *@CommonProps* in der jeweiligen Steuerdatentabelle die Properties der Elemente festgelegt werden, welche gemeinsam auf der Ebene der Planungsgruppe bearbeitet werden können. Eine Änderung auf Ebene der Planungsgruppe wird dann an alle Elemente der Gruppe weitergeleitet, welche die betreffende Property besitzen.

Es gibt eine ganze Reihe begleitender Optionen, die die Verarbeitung der gemeinsamen Eigenschaften beeinflussen. Diese sind im Abschn. 6.1 der Application Note AN-2006-01 [an0601] beschrieben. Auf einige davon wird auch in den folgenden (Unter-)Abschnitten eingegangen.

Darüber hinaus – ggf. zusätzlich zu den via *@CommonProps* generierten Properties – können in abgeleiteten, projekt-spezifischen Klassen mittels der OFML-Schnittstelle *Property* [property] auch eigene Properties programmiert werden, s. Abschn. 5.9.4.

5.9.1 Erzeugen und Aktualisieren der *CommonProps*

Für welche von den in der Option *@CommonProps* spezifizierten Properties tatsächlich ein Gruppenmerkmal angelegt wird, hängt von den Gruppenelementen ab, die bei der Generierung der Gruppenmerkmale herangezogen werden, und von deren aktueller Konfiguration. Dies kann mit Hilfe folgender Optionen gesteuert bzw. beeinflusst werden:

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

Die initiale Erzeugung der gemeinsamen Eigenschaften erfolgt während der Initialisierung der Gruppeninstanz nach der Erzeugung der initialen Layout-Elemente[43].

Eine automatische **Aktualisierung** (Neu-Erzeugung) findet bei folgenden Ereignissen statt:

1. Bei Änderung einer gemeinsamen Eigenschaft, wenn sich diese auf mindestens ein Gruppenelement ausgewirkt hat[44].
   Damit wird auf mögliche Abhängigkeiten zwischen den gemeinsamen Eigenschaften reagiert.

[43] in der Regel in der Methode *setArticleSpec()*
[44] via Methode *fixPropsChanged()*

2. Beim Öffnen und Aktualisieren des Eigenschaftseditors für eine OFML-Instanz – via Methode *updateProperties()* der OFML-Schnittstelle *Property*[45] – wenn sich seit dem letzten Öffnen bzw. Aktualisieren die für Produktdaten der Serie zu verwendende Sprache geändert hat. Damit wird auf eine geänderte Spracheinstellung seitens des Anwenders reagiert.
   Ab XOI 1.60 (mit den Herbst-Releases 2023) erfolgt die Aktualisierung generell auch während des ersten Aufrufs von *updateProperties()* nach Aufruf von *setAddStateCode()*[46]. Damit wird auf mögliche Änderungen in den Produktdaten reagiert, z.B. geänderte Auswahllisten in den zugrunde liegenden Properties.
3. Falls die Option *@AllObjs4CommonProps* den Wert `1` hat:
   Beim Erzeugen eines Gruppenelementes mittels *CreateObj*-Aktion oder mittels globaler Funktion *xOiCreateArticle()* im Modus *@extended* (s. Abschn. 4.1) und beim Entfernen eines Gruppenelementes[47].
   Damit werden Properties erfasst, die nur für das neue bzw. zu löschende Element relevant sind.

In bestimmten Situationen, die nicht durch die oben genannten Standard-Ereignisse abgedeckt sind, muss die Aktualisierung aus den OFML- bzw. OAP-Daten heraus angestoßen werden. Dazu ist in den XOI-Klassen für alle Gruppentypen die Methode ***updateCommonProperties()*** implementiert (s.a. Dokument [methods]). Typische Anwendungsfälle sind:

- Es wurde eine Property geändert, die sich auf die Sichtbarkeit oder den Zustand von gemeinsamen Eigenschaften auswirkt, wobei die geänderte Eigenschaft selber nicht in der Menge der gemeinsamen Eigenschaften enthalten ist.
- Es wird ein Gruppenelement erzeugt bzw. entfernt[48], das Properties besitzt, die in den gemeinsamen Eigenschaften genannt sind, aber das Erzeugen bzw. Entfernen wird nicht durch die oben beschriebene Standard-Situation 3 erfasst.

Der Aufruf der Methode kann und sollte direkt in den OAP-Daten mittels einer entsprechenden *MethodCall*-Aktion nach den betreffenden Aktionen bewerkstelligt werden. (Eine spezielle Programmierung ist dafür also nicht erforderlich.) Wird ein Element durch eine Aktion entfernt, die durch einen Interaktor des Elementes selber ausgelöst wird, ist das jedoch nicht möglich (s. Abschn. 4.4). In dem Fall sollte überlegt werden, das Entfernen des Elementes über einen Interaktor am übergeordneten Artikel zu realisieren.

5.9.2 Gemeinsame Eigenschaften für neues Element

Beim Einfügen eines neuen Elementes in die Planungsgruppe wird das Element in der durch die betreffende Aktion definierten Konfiguration eingefügt (insofern keine Metadaten-gesteuerte Vererbung von Merkmalswerten stattfindet). Diese Konfiguration kann von den aktuellen Einstellungen der gemeinsamen Eigenschaften der Planungsgruppe abweichen. Sollen die aktuellen Einstellungen der gemeinsamen Eigenschaften der Planungsgruppe für das neue Element übernommen werden, muss in einer zusätzlichen Aktion nach der Element-erzeugenden Aktion auf der Gruppeninstanz die Methode ***assignCommonPropValues()*** gerufen werden[49].

[45] in der überschiebenen Implementierung in der Basisklasse *xOiPlGroup*
[46] im Zuge der Wiederherstellung einer Artikelinstanz anhand einer gespeicherten Warenkorb-Repräsentation
[47] via Ereignistypen *@ArticleInserted* bzw. *@ElementRemoval*
[48] z.B. als Ergebnis einer Property-Änderung
[49] Die Methode ist in allen Klassen gleichermaßen definiert und implementiert. Sie ist im Dokument [methods] beschrieben, ebenso wie die beiden unten genannten Methoden.

(Ob für das neue Element tatsächlich die aktuellen Einstellungen der gemeinsamen Eigenschaften übernommen werden sollen, oder doch eher eine Metadaten-gesteuerte Vererbung von Merkmalswerten des Referenzobjektes erfolgen soll, muss projekt- bzw. situationsspezifisch entschieden werden[50].)

Die Methode *assignCommonPropValues()* erwartet als Argument das Element, auf welches das Verfahren angewendet werden soll, in dem betrachteten Szenario also das mittels der *CreateObj*-Aktion erzeugte Element. Dieses wird am besten mit Hilfe der Objektkategorie *MethodCall* spezifiziert, wobei die entsprechenden Methoden zum Zugriff auf das Nachbarelement des durch die *CreateObj*-Aktion definierten Referenzobjektes zum Einsatz kommen (s.a. Abschn. 5.1).

Wird die *CreateObj*-Aktion z.B. im Rahmen eines *Add*-Interaktors ausgeführt, der an das selektierte Gruppenelement (aktives Objekt) gebunden ist, und ist das aktive Objekt gleichzeitig auch das Referenzobjekt für die *CreateObj*-Aktion, so könnten die Methoden *neighbor()* (*xOiJointPlGroup*) bzw. *getNeighbor()* (*xOiLayoutGroup* und *xOiCustomPlGroup*) verwendet werden, wobei im ersten Argument mittels des Platzhalters `$SELF` das aktive Objekt übergeben wird und im zweiten Argument die Anfügerichtung (*xOiJointPlGroup*) bzw. der bei der *CreateObj*-Aktion verwendete Anfügepunkt (*xOiLayoutGroup* und *xOiCustomPlGroup*).

5.9.3 Merkmalsgruppen

Standardmäßig werden bei der Generierung der gemeinsamen Eigenschaften (Option *@CommonProps*) nicht die Property-Klassen von den Elementen übernommen und es wird auch keine spezifische Klasse zugewiesen. Im Eigenschaftseditor erscheinen die generierten gemeinsamen Eigenschaften somit in der Standardgruppe, also „Artikel“ oder „Sonstige“, wenn zusätzliche programmierte Properties (s. nächster Abschn.) einer eigenen Klasse zugewiesen sind.

In den meisten Projekten ist dieses Verhalten ausreichend. In folgenden Situationen kann jedoch auch eine Klassenzuweisung für die generierten gemeinsamen Eigenschaften sinnvoll bzw. sogar notwendig sein:

1. Bei einer großen Menge an gemeinsamen Eigenschaften kann mit Klassen/Gruppen die Übersichtlichkeit für den Anwender erhöht werden.
2. Werden die gemeinsamen Eigenschaften von unterschiedlichen Objekttypen gezogen, und besitzen diese jeweils disjunkte Mengen relevanter Properties:
    a) macht es Sinn, dies auch dem Anwender sichtbar zu machen[51]
    b) sind je eigene Merkmalsgruppen notwendig, wenn (verschiedene) Properties der unterschiedlichen Objekttypen dieselbe sprachspezifische Bezeichnung haben[52]

Im einfachsten Fall erfolgt die Klassenzuweisung für die generierten gemeinsamen Eigenschaften durch Setzen des Wertes `1` für die Option ***@Classes4CommonProps***[53]. Aktuell werden dabei für alle generierten gemeinsamen Eigenschaften die Klassen von den Gruppenelementen übernommen. Wenn dies nicht gewünscht ist, oder wenn im Zusammenhang mit zusätzlichen programmierten Properties (s. nächster Abschn.) spezifische Klassen zugewiesen werden sollen, muss dies in abgeleiteten Klassen mittels der Methode *setPropClass()* der OFML-Schnittstelle *Property* [property] erfolgen.

[50] Merkmalswerte des Referenzobjektes können von den gemeinsamen Eigenschaften abweichen, wenn die Eigenschaften des Referenzobjektes nach der letzten Änderung von gemeinsamen Eigenschaften noch einmal separat geändert wurden.
[51] Für die Anwender ist dann leichter ersichtlich bzw. verständlich, dass sich die Änderung einer Property aus einer gegebenen Merkmalsgruppe auch nur auf bestimmte Elemente der Planungsgruppe auswirkt.
[52] und somit nicht durch die Anwender unterschieden werden können
[53] Voraussetzung ist hier natürlich, dass in den OFML-Daten der Gruppenelemente auch geeignete Property-Klassen angelegt sind.

In beiden Fällen sollten bzw. müssen in der Serie des Planungsgruppenartikels zu den Namen der Property-Klassen dann noch entsprechende sprachspezifische Text-Ressourcen (`.sr`-Dateien) angelegt werden.

5.9.4 Programmierte Gruppenmerkmale

In vielen Projekten sollen die Anwender die Möglichkeit haben, allgemeine Planungseigenschaften der Gruppe beeinflussen zu können, z.B. die erlaubten maximalen Ausmaße. Dazu müssen in der projekt-spezifisch abgeleiteten Gruppenklasse entsprechende Properties programmiert werden. In der Regel erfolgt das dann zusätzlich zu den automatisch generierten gemeinsamen Eigenschaften (Option *@CommonProps*).

Gelegentlich müssen in der projekt-spezifischen Klasse auch Properties programmiert werden, weil die Properties der Gruppenelemente nicht genug Funktionalität bereitstellen bzw. diese nicht eins-zu-eins für die Gruppe übernommen werden können[54].

Im Folgenden einige Hinweise und Tipps zur Programmierung von Gruppenmerkmalen.

1. Da die Liste der Schlüssel der programmierten Properties an mehreren Stellen der Implementierung der Gruppenklasse benötigt wird, empfiehlt sich die Definition einer entsprechenden statischen Klassenvariablen, z.B.:

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

2. Wie in der Application Note AN-2006-01 schon bei der Option *@CommonProps* beschrieben, sollten für eine bessere Performanz auch die programmierten Properties in der Tabelle `non_pd_properties` angegeben werden.

3. Die Basisklasse *xOiPlGroup* für alle Planungsgruppentypen implementiert einige Standardbehandlungen (u.a. die Persistenz betreffend) für die Properties einer Gruppeninstanz, deren Schlüssel durch die Hakenmethode ***getFixProperties()*** geliefert wird. In der Regel sollte also diese Methode in der projekt-spezifischen Gruppenklasse überschrieben werden und die Schlüssel der programmierten Properties liefern, ggf. zusätzlich zu den von der geerbten Implementierung gelieferten Schlüsseln der generierten gemeinsamen Eigenschaften.

   Beispiel siehe Abschnitt 5.10.1.

4. Für die Definition der Properties werden die Methoden der neuen *Property*-Schnittstelle [property] empfohlen. (Die für die Bezeichnungen der Property und der ggf. vorhandenen Choicelist-Werte zu verwendende Sprache wird mittels der Methode *getPDLanguage()* ermittelt[55].)

5. Werden die Properties zusätzlich zu den automatisch generierten gemeinsamen Eigenschaften (*@CommonProps*) angelegt, muss entschieden werden, ob die programmierten Properties im Eigenschaftseditor ***vor*** oder ***nach*** den generierten gemeinsamen Eigenschaften angezeigt werden sollen. Standardmäßig werden die generierten gemeinsamen Eigenschaften in der Property-Liste ab der Position 1 angelegt[56].

[54] Dies ist oft der Fall, wenn das OAP-Projekt auf bestehende OFML-Daten aufgesetzt wird. In Projekten, wo die OFML- und die OAP-Daten zusammen (neu) aufgebaut werden, sollte von vornherein darauf geachtet werden, dass in den OFML-Daten der Gruppenelemente die Properties schon so angelegt werden, dass eine spezifische Programmierung von Properties in der Gruppenklasse vermieden wird.
[55] Details siehe [property]
[56] in der Reihenfolge gemäß Option *@CommonProps*

- Sollen die programmierten Properties danach folgen, muss ihnen eine entsprechend hohe Positionsnummer zugewiesen werden.
- Im anderen Fall muss die Option *@FirstPos4CommonProps* verwendet werden, um eine genügend hohe Positionsnummer für die erste generierte gemeinsame Eigenschaft festzulegen.

Mit Hilfe der Option ***@CommonPropsPos***[57] kann jedoch auch eine gemischte Reihenfolge realisiert werden!

6. Die **initiale Definition** der programmierten Properties erfolgt in der überschriebenen Methode *setArticleSpec()* nach Aufruf der geerbten Implementierung und ggf. nach Erzeugung zusätzlicher Start-Elemente (s. Abschn. 5.7.).

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

    initMyProps();
}
```

Die Zuweisung der initialen Werte erfolgt zur Vermeidung von Overhead (Performanz) und von Nebeneffekten *nicht* mittels *setPropValue()*, sondern durch Setzen des Wertes in der Tabelle der dynamischen Merkmale (s. Methode *getDynamicProps()* in der OFML-Schnittstelle *Base* [ofml]) bzw. durch Aufruf der entsprechenden *set*-Methoden.

7. Die Behandlung einer **Änderung** einer programmierten Property durch die Anwender erfolgt in der überschriebenen Methode *fixPropsChanged()*, ggf. nach Aufruf der geerbten Implementierung, welche die Standardbehandlung für die generierten gemeinsamen Eigenschaften vornimmt (s.a. Abschn. 5.9.1):

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

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

        // handle change of my property tP
        ...
    }

    return(tRet);
}
```

8. Wenn durch die Anwender zur Laufzeit in der Applikation die **Spracheinstellung geändert** wird und zu diesem Zeitpunkt die Gruppeninstanz selektiert ist (geöffneter Eigenschaftseditor), müssen die sprachspezifischen Bezeichnungen der programmierten Properties inklusive ihrer Werte und Klassen geändert werden[58].

[57] Ab XOI 1.60 (Herbst 2023)
[58] Das gilt auch, wenn eine gespeicherte Gruppeninstanz zum (Nach)Konfigurieren geöffnet wird, und in der Applikation eine andere Sprache eingestellt ist, als zum Zeitpunkt des Speicherns.

Dies erfolgt durch Überschreiben der Methode *updateOtherFixProperties2()*[59]. Diese Hakenmethode wird von der Standardimplementierung der Methode *updateProperties()* der Basisklasse *xOiPlGroup* gerufen, wenn sich seit dem letzten Aufruf die für die Produktdaten der Serie zu verwendende Sprache geändert hat (s.a. Abschn.5.9.1, Punkt 2). Zu diesem Zeitpunkt wurden in der Basisklasse schon die Bezeichnungen für die generierten gemeinsamen Eigenschaften geändert. Diese müssen in *updateOtherFixProperties2()* also nicht mehr behandelt werden[60].

Wenn die programmierten Properties mit Hilfe der Methode *setupProperty2()* der *Property*-Schnittstelle erzeugt werden (siehe Punkt 4 oben), reicht zum Anpassen der Texte ein Aufruf von *setPropName()* und ggf. *setPropChoiceList()* (falls keine Text-Ressourcen für die Werte verwendet werden). Falls die Properties mit Hilfe der Methode *setupProperty()* der alten *Property*-Schnittstelle erzeugt werden, müssen diese neu mit dieser Methode erzeugt werden. In beiden Fällen wird die nun zu verwendende Sprache mittels der Methode *getPDLanguage()* ermittelt[61].

9. Wenn bei der Definition von programmierten Properties die Auswahllisten und ggf. weitere Attribute von Properties von Gruppenelementen herangezogen werden, muss auf mögliche Änderungen in den zugrunde liegenden Produktdaten reagiert werden.
   Dies erfolgt ebenfalls durch Überschreiben der Methode *updateOtherFixProperties2()*[62]. Diese Hakenmethode wird, zusätzlich zu den im Punkt 8 beschriebenen Situationen, von der Standardimplementierung der Methode *updateProperties()* der Basisklasse *xOiPlGroup* gerufen, wenn es sich um den ersten Aufruf nach dem Aufruf von *setAddStateCode()* handelt (s.a. Abschn.5.9.1, Punkt 2). Zu diesem Zeitpunkt wurden in der Basisklasse schon die gemeinsamen Eigenschaften neu erzeugt[63].
   Im Gegensatz zur Situation aus Punkt 8 – die Methode *updateOtherFixProperties2()* besitzt Parameter, anhand derer die auslösende Situation erkannt werden kann – ist hier die einfachste und sicherste Lösung vermutlich, die betreffenden programmierten Properties komplett neu zu definieren (s. Punkt 6).

5.10 Persistenz

Die XOI-Basisklassen für Planungsgruppen speichern alle Informationen, die zum Nachkonfigurieren der Planungsgruppe notwendig sind. Dies erfolgt im Wesentlichen mittels der Methoden *getAddStateCode()* und *setAddStateCode()* der OFML-Schnittstelle *Article* [article]. Bei Erweiterungen in abgeleiteten Klassen müssen folgende Aspekte berücksichtigt werden, damit das geerbte Verhalten weiter funktioniert bzw. um ein Überschreiben der beiden genannten Methoden zu vermeiden.

5.10.1 Properties

Werden in der abgeleiteten Klasse eigene Properties programmiert, müssen die Keys dieser Properties durch die überschriebene Methode *getFixProperties()* geliefert werden. Werden diese Properties zusätzlich zu den Properties definiert, die durch die Basisklasse gemäß Option *@CommonProps* in der Steuerdatentabelle generiert werden, muss dies entsprechend bei der Implementierung berücksichtigt werden:

[59] Ab XOI 1.60 (Herbst 2023). Davor wurde die Hakenmethode *updateOtherFixProperties()* verwendet. Diese wird aus Gründen der Abwärtskompatibilität zwar weiter unterstützt, gilt aber als veraltet.
[60] Deswegen das „Other“ im Methodennamen.
[61] Details siehe [property]
[62] Ab XOI 1.60 (Herbst 2023)
[63] Diese müssen in *updateOtherFixProperties2()* also nicht mehr behandelt werden.

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

    xOiCopyAggr(sMyPropKeys, tRet, 0);

    return(tRet);
}
```

5.10.2 Membervariablen

Werden in Membervariablen Informationen gespeichert, die bei einer Nachkonfiguration benötigt werden, und können diese nicht anderweitig wieder hergestellt werden, müssen die Werte dieser Membervariablen im sogenannten *AddStateCode* kodiert werden (*getAddStateCode()*) und bei der Wiederherstellung aus diesem wieder ausgelesen und zugewiesen werden (*setAddStateCode()*).

Um das relativ aufwändige Überschreiben der beiden Methoden zu vermeiden, bieten alle XOI-Basisklassen für Planungsgruppen einen Mechanismus an, der auf der Methode ***getAddStateMembers()*** beruht: Für alle Membervariablen, deren Namen (*String*) durch die Methode geliefert werden (Rückgabewert ist vom Typ *Vector*), übernimmt die XOI-Basisklasse die Kodierung im *AddStateCode* und die Wiederherstellung aus diesem.

Unabhängig davon, ob die Kodierung via *getAddStateMembers()* erfolgt, oder nicht, ist zu beachten, dass die zu kodierenden Membervariablen **keine Objektreferenzen** enthalten dürfen!

Es gibt 2 Ansätze, diese Limitierung zu umgehen:

1. Die beiden o.g. Methoden werden überschrieben: in *getAddStateCode()* erfolgt vor dem Aufruf der geerbten Implementierung eine Konvertierung der Objektreferenzen in eine speicherfähige Information (z.B. Objektname, s.u.) und in *setAddStateCode()* werden entsprechend nach Aufruf der geerbten Implementierung die Objektreferenzen wieder hergestellt.
2. In den betreffenden Membervariablen werden keine Objektreferenzen gespeichert, sondern eine Information, aus der bei Bedarf die jeweilige Objektreferenz ermittelt werden kann (z.B. Objektname, s.u.).

Achtung: Bei Verwendung von Objektnamen darf nicht der vollständige (hierarchische) Objektname verwendet werden, sondern nur der lokale Name innerhalb der Planungsgruppe!
Zur Realisierung dieser Ansätze stehen in allen XOI-Basisklassen für Planungsgruppen die Methoden *localObjName()* und *localName2Obj()* zur Verfügung. Diese sind in der Basisklasse *xOiPlGroup* spezifiziert und implementiert.

Alternativ zum (lokalen) Objektnamen kann der Index der Instanz gespeichert werden, unter dem die Instanz in der Liste der Elemente der Gruppeninstanz geführt wird (siehe Methode *getElements()* der OFML-Schnittstelle *MObject* [ofml]).
Muss eine Liste von Objektreferenzen gespeichert werden, können dazu die globalen XOI-Funktionen *xOiElRefs2Elldcs()* und *xOiElldcs2ElRefs()* verwendet werden[64]. Ein Beispiel zur Verwendung der Methoden siehe unten.

Objektreferenzen in Membervariablen müssen (aktuell) auch noch in den OFML-**Persistenzregeln** beachtet und behandelt werden[65]!

[64] s. Sektion „xOiFuncs“, Abschn. „DUMP and EVAL rule supporting functions“ in der XOI-Dokumentation [xoi]
[65] Die in den Applikationen verwendete Warenkorb-Implementierung (OBK) schreibt beim Löschen und Kopieren eines Artikels einen Dump der Artikelinstanz in den Cut-Buffer.

Angenommen, es gibt eine Membervariable *mMyEL*, welche eine Liste von Objektreferenzen enthält, könnten die Persistenzregeln wie folgt implementiert werden:

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

    return(0);
}

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

    return(0);
}

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

    return(0);
}
```

5.10.3 Zusätzliche Startelemente

Gemäß Abschnitt 5.7 können in überschriebenen Implementierungen der Methode *setArticleSpec()* weitere Elemente erzeugt werden, zusätzlich zu den Start-Elementen, die in der geerbten Implementierung entsprechend der Optionen *@StartElement* bzw. *@StartLayout* erzeugt werden.

Damit die Wiederherstellung durch die Methode *setAddStateCode()* der Basisklasse funktioniert, muss folgendes beachtet werden:

- Bei Ableitung von *xOiJointPlGroup* und *xOiLayoutGroup* muss sichergestellt werden, dass beim Überschreiben der Methode *elRemoveValid()* ein Entfernen von Elementen, die während *setArticleSpec()* erzeugt wurden, nicht verhindert wird.
- Bei Ableitung von *xOiTabularPlGroup* und *xOiCustomPlGroup* dürfen in *setArticleSpec()* keine zusätzlichen Elemente erzeugt werden, wenn der Aufruf im Rahmen der Wiederherstellung der Planungsgruppe zur Nachkonfiguration erfolgt. Die Unterscheidung erfolgt anhand der Methode *xOiBasePlanning::getCreationMode()*[66]:

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

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

[66] Details siehe XOI-Dokumentation [xoi], Sektion „Main Planning Classes“

5.11 Überschreiben von Methoden der XOI-Basisklassen

Gelegentlich müssen vor oder nach dem Aufruf einer Methode der Gruppeninstanz via OAP-Aktion vom Typ *MethodCall* weitere Behandlungen durchgeführt werden. Idealerweise sollte das durch zusätzliche, vor- bzw. nachgelagerte OAP-Aktionen erledigt werden[67]. Ist dies aus bestimmten Gründen nicht möglich, oder ist es effizienter, alles in einem Methodenaufruf zu erledigen, sollte die betreffende Methode in der abgeleiteten Klasse nicht direkt überschrieben werden (um dort die zusätzlichen Behandlungen zu implementieren), da dies die Weiterentwicklung der verwendeten XOI-Basisklasse behindern kann[68]. Stattdessen sollte eine eigene Methode definiert und implementiert werden, in der dann an geeigneter Stelle die betreffende Methode der XOI-Basisklasse gerufen wird!

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

    xoiMethod(...);

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

5.12 Verwaltung des Property-Zustands von Gruppenelementen

Oft besteht die Aufgabenstellung, dass bestimmte, normalerweise editierbare Properties einer Artikelinstanz nicht editierbar (oder gar nicht sichtbar) sein sollen, wenn die Artikelinstanz ein Element einer Planungsgruppe ist. Zum Beispiel kann gefordert sein, dass eine Material-Property, z.B. *@Color*, die mittels Option *@CommonProps* auf die Ebene der Planungsgruppe hochgezogen wird (s. Abschn. 5.9), für die Elemente der Gruppe nicht editierbar ist, um eine einheitliche Ausführung der Planungsgruppe zu gewährleisten.

Hinweis:
Wird eine Property der Gruppenelemente mit den in diesem Abschnitt beschriebenen Lösungsansätzen nicht editierbar oder nicht sichtbar geschaltet, soll aber als Bestandteil der *@CommonProps* auf der Ebene der Planungsgruppe sicht- und editierbar sein, muss entsprechend die Option *@ROPropsEditable4Common* bzw. die Option *@NonVisibleProps4Common* genutzt werden!

Ein Lösungsansatz dafür umfasst folgende Schritte:

- Für die Start-Elemente der Planungsgruppe wird in der Option *@StartLayout* (bzw. *@StartElement*) der Status der betreffenden Properties mit spezifiziert, z.B.:

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

  Der Property-Status muss an dieser Stelle gemäß der Methode *setPropState2()* der OFML-Schnittstelle *Property* [property] angegeben werden. Der Wert `1` spezifiziert dabei eine sichtbare, aber nicht editierbare Property.

[67] gemäß dem Grundsatz der Vermeidung von OFML-Programmierung
[68] z.B. die Einführung zusätzlicher optionaler Parameter

- Für nachträglich via OAP-Interaktor in die Planungsgruppe eingefügte Elemente wird in der Aktionsliste des Interaktors (bzw. der betreffenden Einträge einer *ActionChoice*) nach der *CreateObj*-Aktion noch eine *PropChange*-Aktion spezifiziert (s.a. Abschnitt 5.1), z.B.:

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

Man beachte, dass dieser Lösungsansatz ohne zusätzliche OFML-Programmierung auskommt. Leider **funktioniert** dieser Ansatz **nicht**, wenn es sich bei der betreffenden Property um ein originäres, **kaufmännisches Merkmal** des Artikels handelt und dieser durch eine **Metatyp**-Instanz gekapselt wird: nach einer durch die Planungsgruppe ausgelösten Property-Änderung hat die Property wieder ihren ursprünglichen Zustand (editierbar).

Der oben beschriebene Ansatz muss dann folgendermaßen modifiziert und erweitert werden:

- Die betreffende kaufmännische Property wird als sogenanntes *na-Merkmal* auf die Ebene der Metatyp-Instanz hochgezogen. (Der Name der Property ändert sich dann standardmäßig zu *@GColor* und muss entsprechend auch an den oben genannten Stellen geändert werden.)

- Für die Start-Elemente muss in einer spezifisch abgeleiteten Gruppensklasse die Methode *setArticleSpec()* nach folgendem Muster (für eine *xOiJointPlGroup*) überschrieben werden:

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

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

- Für nachträglich via OAP-Interaktor in die Planungsgruppe eingefügte Elemente muss in der betreffenden Aktionsliste nach der *CreateObj*-Aktion und der oben beschriebenen *PropChange*-Aktion noch eine *MethodCall*-Aktion nach folgendem Muster spezifiziert werden:

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

  Anmerkung:
  Die Methode *GoMetaType::updateNAPropState()* gewährleistet eine dauerhafte Änderung des Status eines na-Merkmals. Der Property-Status muss hier gemäß der Methode *setPropState()* der alten OFML-Schnittstelle *Property* angegeben werden, d.h., eine sichtbare, aber nicht editierbare Property wird hier durch den Wert `0` spezifiziert.

Bei **Metatyp-basierten Artikeln** ist unter Umständen folgender Lösungsansatz **besser** geeignet, insbesondere dann, wenn der Zustand von mehreren (vielen) Properties modifiziert werden muss[69]. Bei Metatypen ist das oft der Fall, da neben dem o.g. Beispiel mit dem Material-Merkmal oft auch Merkmale zum Artikeltausch im Kontext einer Planungsgruppe nicht sinnvoll sind und deaktiviert werden müssen:

- Im Meta-Typ wird ein (nicht sichtbares) Hilfsmerkmal angelegt, z.B. *In_PGroup*, mit den möglichen Werten `0` (nein) und `1` (ja), wobei `0` der initiale Wert ist.
- In der Tabelle `go_actions` werden mittels einer CON_PROP-Aktion die betreffenden Merkmale[70] deaktiviert, wenn die Bedingung `In_PGroup == 1` erfüllt ist.

[69] Die Option *@StartLayout* sowie die Aktionslisten würden nach dem ersten Ansatz dann sehr lang und unübersichtlich.
[70] Kaufm. Merkmale wie *@Color* müssen dazu auch, wie im ersten Ansatz, als na-Merkmale angelegt werden.

- Im OAP wird das Merkmal *In_PGroup* auf `1` gesetzt. (Für die Start-Elemente erfolgt das in der entsprechenden Option der Steuerdaten-Tabelle, für zusätzliche via Interaktor eingefügte Elemente mittels *PropChange*-Aktion nach der *CreateObj*-Aktion in der betreffenden Aktionsliste.)

5.13 Reaktion auf Merkmalsänderungen von Elementen

Oft muss die Gruppeninstanz auf Änderungen bestimmter Properties von Gruppenelementen reagieren. Das kann prinzipiell mit folgenden beiden Ansätzen realisiert werden:

1. In den betreffenden Klassen der Gruppenelemente wird die Methode *setPropValue()* oder *propsChanged()* überschrieben und bei Änderung einer relevanten Property an eine entsprechende Methode der übergeordneten Gruppeninstanz delegiert.
2. Die Gruppeninstanz registriert sich beim globalen **Change Manager** (Typ *OiChangeManager*) für Änderungsereignisse von Typ *@PropertyChange*, ausgelöst von Elementen der Gruppe, und reagiert darauf in der Methode *receivePropertyChange()* (Beispiel s.u.)[71].
   (Bei kaufmännischen Properties, die als sogenannte *na-Merkmale* auf die Ebene einer kapselnden Metatyp-Instanz hochgezogen werden, muss ggf. der Ereignistyp *@MetaMainChildPropChange* verwendet werden[72].)

In den meisten Fällen ist der 2. Ansatz vorzuziehen, da dieser keine Implementierungen in projekt-spezifischen Elementklassen erfordert[73]!

Außerdem erleichtert der 2. Ansatz die Lösung für folgende Problemstellung:

Enthält die Gruppe als Elemente **Metatyp-Instanzen**, werden während der (Wieder)Herstellung der Gruppe zur Nachkonfiguration von der Basisklasse *GoMetaType* bei der Verarbeitung des gespeicherten Varianten-Codes Property-Änderungen ausgelöst. Für die Gruppeninstanz sind diese Änderungen irrelevant, da die Gruppe am Ende des Vorgangs der (Wieder)Herstellung den korrekten, gespeicherten Zustand besitzt. Schon aus Gründen der Performanz sollte die Gruppeninstanz also nicht auf diese Änderungen reagieren. Mehr noch, in bestimmten Fällen kann es zu Fehlverhalten bzw. Evaluierungsfehlern kommen, wenn die Gruppe darauf reagiert, da zu diesem Zeitpunkt noch nicht alle (anderen) Gruppenelemente komplett wieder hergestellt sind!

Nachfolgend das Implementationsmuster auf der Basis des 2. Ansatzes:

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

    var tChMgr = oiGetChangeManager();

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

[71] Das Konzept des globalen Change Managers und die aktuell unterstützten Ereignistypen sind in der XOI-Dokumentation [xoi] in der Sektion „ChangeManager Event Types“ beschrieben.
[72] Bei diesen Properties wird die Property-Änderung an das sogenannte Hauptkind der Metatyp-Instanz delegiert. Die Methode *setPropValue()* für das Hauptkind meldet zwar (auch) ein Ereignis vom Typ *@PropertyChange*, zu dem Zeitpunkt fanden aber noch keine, evtl. erforderlichen/relevanten Anpassungen in der Metatyp-Instanz selber statt.
[73] die unter Umständen auch erst extra für diesen Zweck von den Standardklassen abgeleitet werden müssten

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

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

5.14 Kollisionserkennung

Wird mittels *CreateObj*-Aktion oder der Funktion *xOiCreateArticle()* (bzw. *xOiCreateArticle2()*) ein neues Element an einen definierten Anfügepunkt eines bereits in der Gruppe befindlichen Elementes angefügt[74], so findet standardmäßig eine Prüfung statt, ob das neue Element mit anderen Elementen der Gruppe kollidiert[75]. Wird eine Kollision erkannt, wird das Einfügen des neuen Elementes abgewiesen.

Bei ungenauen Geometriedaten bzw. ungenau definierten Anfügepunkten wird in den OFML-Daten oft die Kollisionserkennung für die Gruppenelemente ganz oder temporär (während des Einfüge-Vorgangs) abgeschaltet[76]. Das hat folgende Konsequenzen:

1. Bei „um die Ecke“ geplanten Gruppen kann das neue Element mit anderen Gruppenelementen kollidieren.
2. Das Einfügen des neuen Elementes *zwischen* zwei andere Gruppenelemente (*InsertMode* `2`) ist nicht möglich!

Um diese Probleme zu vermeiden, sollten also die Geometriedaten und Anfügepunkte von vorn herein möglichst genau definiert werden, so dass auf das Abschalten der Kollisionserkennung verzichtet werden kann!

Kann die Abschaltung der Kollisionserkennung nicht verhindert werden, kann zumindest das zweite Problem (bei Verwendung von *xOiCreateArticle()* bzw. *xOiCreateArticle2()*) mit etwas Programmieraufwand umgangen werden. Die Grundidee besteht dabei darin, mittels der Methoden *isBusyAttPt()*[77] bzw. *isFreeAttPt()*[78] zu prüfen, ob der Anfügepunkt des Referenzobjektes schon belegt ist, und in diesem Fall die Kollisionserkennung zu erzwingen.

Das folgende Implementationsmuster behandelt den Fall, dass bei einer ***Metatyp*-basierten Datenanlage** die Kollisionserkennung mittels *GSetup*-Flag `4096` temporär für das neue Element deaktiviert wird.

In der von *GoMetaType* projekt-spezifisch abgeleiteten Klasse muss die Methode *isEnabledCD()* überschrieben werden:

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

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

    return(tRes);
}
```

[74] Betrifft die Gruppentypen *xOiCustomPlGroup*, *xOiJointPlGroup* und *xOiLayoutGroup*
[75] via Methode *checkChildColl()* der OFML-Schnittstelle *Complex* [ofml]
[76] via Methode *disableCD()* der OFML-Schnittstelle *Base* [ofml]
[77] Klasse *xOiJointPlGroup*
[78] Klassen *xOiLayoutGroup* und *xOiCustomPlGroup*

In der projekt-spezifischen Klasse für die Planungsgruppe *myLayoutGroup*:

```
static var sNeedCD4Insertion = 0;

func needCD4Insertion()
{
    return(sNeedCD4Insertion);
}

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

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

    sNeedCD4Insertion = 0;
    ...
}
```

Das obige Implementationsmuster setzt voraus, dass die Kollisionserkennung für die Kind-Objekte der Metatyp-Instanz (defaultmäßig) eingeschaltet ist[79]!

Die **allgemeinere Lösung** der Problematik basiert also nicht auf dem Überschreiben von *isEnabledCD()* in den Klassen der Gruppenelemente, sondern auf dem temporären Aktivieren der Kollisionserkennung *während xOiCreateArticle()* (bzw. *xOiCreateArticle2()*):

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

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

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

    // tNewEl.disableCD(); // ?

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

    ...
}
```

[79] Liefert *isEnabledCD()* für die Metatyp-Instanz 1, findet zwar die Kollisionsprüfung für deren Kind-Objekte mittels der globalen Funktion *oiCollision()* statt, dabei gilt aber dann der Status, der via *disableCD()* bzw. *enableCD()* für die gegeneinander zu prüfenden Objekte definiert ist.

6 OFML-Debugging

Die Ausführungen in diesem Abschnitt beziehen sich auf das OFML-Debugging im pCon.planner mittels der OFML-Konsole, die mit dem Plugin X3G-OFC bereitgestellt wird.

6.1 Debuggen des *xOiOAPManager*

Der *OAP-Manager* ist die Schnittstelle zwischen den OAP-Daten und den Applikationen. Zur Laufzeit existiert dazu genau eine Instanz der Klasse *xOiOAPManager*. Bei Problemen bei der Verarbeitung der OAP-Daten kann ein OFML-Debuglog für diese Klasse hilfreich sein[80]. Dabei sind im Wesentlichen 3 Toplevel-Methoden relevant:

- Bei der Selektion eines Artikels werden durch die Applikation die Methoden *getArticleData()* und *getInteractors()* gerufen (in dieser Reihenfolge).
- Bei Aktivierung eines Interaktors wird durch die Applikation für jede Aktion des Interaktors die Methode *getActionData()* gerufen[81].

(Die ausführliche Spezifikation dieser Methoden ist der XOI-Dokumentation [xoi] zu entnehmen[82].)

Ein erneuter Aufruf von *getArticleData()/getInteractors()* erfolgt – bei unveränderter Selektion – auch, wenn der Mauszeiger in einen anderen View des Planner eintritt (da für jeden View die Interaktor-Symbole ermittelt und angezeigt werden müssen). Um unnötig viele Ausgaben im Logfile zu vermeiden, empfiehlt sich also, im 1-View-Modus zu debuggen.

Fehler in den OAP-Tabellen können relativ einfach und schnell gefunden werden, indem im Logfile (`debug.out`) nach folgenden Ausgaben gesucht wird:

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

Dazu müssen mindestens die Debug-Modi `Warn(ing)`, `Func` und `Func2` gesetzt sein und der Functiontrace-Level sollte mindestens auf 5 eingestellt sein (wenn *ActionChoice*-Aktionen involviert sind, mindestens auf 7).

6.2 Debugging bei *CreateObj*

Wenn eine *CreateObj*-Aktion mit vorgegebenem Anfügepunkt (bzw. auch eine *MethodCall*-Aktion unter Verwendung der globalen Funktionen *xOiCreateArticle()* und *xOiCreateArticle2()*) nicht das gewünschte Ergebnis bringt (Objekt wird nicht erzeugt), ist oft die Ursache nicht gleich offensichtlich. Hier 2 Hinweise zur Fehlersuche:

1.

Eine Ursache kann sein, dass an der Stelle des gewählten Anfügepunktes eine Kollision des neuen Objektes mit bereits existierenden Objekten auftreten würde (entweder durch falsch positionierten Anfügepunkt oder durch "ungenaue" Geometrien).

Ob das zutrifft, kann man recht schnell mittels der Debug-Einstellung *Collision* erkennen. Im Logfile finden sich dann Einträge der Art

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

[80] Die OFML-Konsole stellt dazu bereits ein passendes Debug-Profil OAP bereit.
[81] Tritt beim Zugriff auf die OAP-Daten einer Aktion ein Fehler auf, wird die Abarbeitung der Aktionen abgebrochen.
[82] Die Klasse *xOiOAPManager* ist in der Sektion „Utility Classes“ enthalten.

2.

Wenn 1. nicht zutrifft, muss man (leider) etwas tiefer in die Abläufe bei der Objekterzeugung eintauchen. Hierbei ist zunächst wichtig zu wissen, dass die o.g. Aktionen auf dem sogenannten *checkAdd()*-Mechanismus basieren. Im Detail kann man dazu in der OFML-Spezifikation nachlesen. Für den Moment muss man folgendes wissen:

Auf dem Vaterobjekt wird die genannte Methode aufgerufen, um Position und Rotation für das neue Objekt zu ermitteln. Zuvor wird am Referenzobjekt (Nachbar) mittels der Methode *setActiveAttPt()* (Schnittstelle *AttachPts* [xoi]) der in der Aktion spezifizierte Anfügepunkt aktiviert.

Generell muss man also bei Problemen mit *CreateObj* in den Debug-Einstellungen die verwendete Klasse des Vaterobjektes angeben (und ggf. relevante Basisklassen[83]). Im Log muss man dann nach Hinweisen in den Ausgaben von *checkAdd()* suchen.

In einem ersten Schritt empfiehlt es sich aber, nur die Klasse *OiGlobal* anzugeben und im Log nach der globalen Funktion *oiGetPosRot4AttachPts()* zu suchen. Diese wird nämlich von allen Standard-Implementierungen von *checkAdd()* in den Basisklassen aus OI und XOI zur eigentlichen Ermittlung von Position und Rotation für das neue Objekt verwendet.

Eine Ursache für die Nichterzeugung könnte z.B. sein, dass *für den angegebenen Anfügepunkt keine Definition vorliegt*. Dann würde man dies relativ schnell anhand der Ausgabe für die genannte Funktion erkennen:

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

Sollte dieser Fall nicht vorliegen, könnte die Fehlerursache sein, dass *zum gesetzten Anfügepunkt kein passendes Gegenstück vorhanden ist*. Das erkennt man im Log, indem man noch die Klasse *OiFuncs* hinzunimmt. Bei der Ausgabe im Log zur Funktion *oiCalcCheckAttPt()* (1 Level unter oiGetPosRot4AttachPts()) steht dann folgendes:

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

Die Ursache dafür könnte sein, dass zwar in den Metadaten in der Tabelle `go_attpt.csv` die Anfügepunkte definiert sind, sie aber nicht in der Tabelle `go_action.csv` als Gegenstücke definiert sind.

6.3 Verwendung von *xOiOAPManager::setDBMode()*

Normalerweise wird die OAP-Datenbank, auf die gerade zugegriffen wird, offengehalten, bis auf die OAP-Datenbank eines anderen OFML-Programms zugegriffen wird. Für den OAP-Datenanleger ist das etwas hinderlich, da er/sie nicht „live“ Änderungen in den OAP-Daten des gerade bearbeiteten OFML-Programms testen kann.

Die Klasse *xOiOAPManager* bietet die Methode *setDBMode()*, um dieses Standardverhalten (Modus *@KeepOpen*) zu ändern: Mit dem Modus *@CloseOpen* werden OAP-Datenbanken jedes Mal geschlossen und geöffnet, wenn ein Objekt ausgewählt oder eine Aktion ausgeführt wurde.

[83] bei *xOiJointPlGroup* z.B. noch die Basisklassen *xOiLRPlGroup* und *xOiPlGroup*

Wenn man in der Kommando-Zeile der OFML-Konsole den Befehl

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

eingibt, kann man dann Änderungen in den OAP-Daten sofort testen[84].

[84] Die `oap.ebase` muss dabei jedoch neu erstellt werden.

7 Sonstiges

7.1 EBase für Steuerdatentabellen

Wenn die von den XOI-Basisklassen für Planungsgruppen (s. Abschn. 5) verwendeten Steuerdatentabellen in die `ofml.ebase` übernommen werden sollen und dabei eine ältere EBase-Version als 1.2.3[85] verwendet wird, muss folgende Problematik beachtet werden:

Wenn in einem Tabelleneintrag für die Option *@StartElement* bzw. *@StartLayout* ein leerer Variantencode angegeben wird, wie z.B. in

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

dann muss das Feld in doppelte Anführungszeichen (") eingeschlossen und müssen doppelte Anführungszeichen im Feldinhalt durch 2 aufeinanderfolgende doppelte Anführungszeichen ersetzt werden:

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

Eine solche Transformierung ist auch erforderlich, wenn die Artikelnummer oder der Variantencode ein Semikolon enthält[86].

Tipp:

Die Elemente im Vektor nach der Artikelnummer sind optional. Wenn diese nicht relevant sind, wie im obigen Beispiel, sollten sie weggelassen werden, womit sich dann auch die Transformierung erübrigt:

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

7.2 Metatyp-Typ-Mapping

Die Spezifikation macht keine Aussage darüber, welche Serie im Feld 2 der Tabelle *Metatype2Type* anzugeben ist.

Implementiert ist es aktuell so, dass an der Stelle die Serie des Artikels erwartet wird, nicht die Serie, in der der Metatyp angelegt ist!

Eine Konsequenz daraus ist, dass in dem Fall, wo eine OAP-Datenbank für mehrere kaufmännische Serien angelegt wird, und es Artikel aus den verschiedenen Serien gibt, die *einen* Metatyp (ID) verwenden (egal in welcher Serie dieser angelegt ist), für diesen Metatypen mehrere Einträge in der Mapping-Tabelle anlegt werden müssen, z.B.:

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

(Hier wird es in Zukunft evtl. andere/bessere Regelungen geben. In jedem Fall wird eine zukünftige Änderung im OAP-Kern der Applikationen aber ein abwärtskompatibles Fallback beinhalten, so dass nach dem jetzigen Stand angelegte OAP-Daten auch dann noch korrekt verarbeitet werden.)

[85] enthalten in der Version 1.4.0 der OFML-Binaries
[86] Das entspricht den aus den Spezifikationen von OCD und OAP bekannten Festlegungen zur Repräsentation eines Feldes.

Anhang

A.1 Dokumenthistorie

2023-11-07:

- Diverse kleinere Berichtigungen und Verbesserungen.
- Ergänzende Hinweise in Abschn. 3.3 (dynamische Interaktoren) in Bezug auf das Target-Objekt einer Aktion, deren ID als Argument eines *methodCall()*-Ausdrucks in Bedingungen benutzt wird.
- Neuer Abschnitt 3.5 zur Berücksichtigung des Konfigurationskontextes bei der Sichtbarkeit von Interaktoren.
- In Abschnitt 4.1 wird nun die neue globale Funktion *xOiCreateArticle2()* erwähnt.
- Im Abschn. 5 wird nun die neue Klasse *xOiCustomPlGroup* berücksichtigt.
- Präzisierung zum erweiterten Verhalten von *xOiPlGroup::updateProperties()* im Abschn. 5.9.1
- Präzisierung zur Verwendung der Methode *getPDLanguage()* im Abschn. 5.9.4.
- Neuer Hinweis im Abschn. 5.9.4 auf neue Option *@CommonPropsPos*.
- Neuer Hinweis im Abschn. 5.9.4 zur Reaktion auf mögliche Änderungen in den Produktdaten, wenn Definitionen von programmierten Properties aus Attributen (z.B. Auswahllisten) von Properties von Gruppenelementen abgeleitet werden.
- Neuer Hinweis auf den Ereignistyp *@MetaMainChildPropChange* im Abschn. 5.13.
- Zusätzliche Hinweise zur Kollisionserkennung im Abschn. 5.14.
- Neuer Abschnitt 6.3.

2022-01-03:

- Neuer Abschnitt 5.14 mit Hinweisen zur Kollisionserkennung.
- Korrektur und Ergänzung zu den Debug-Einstellungen im Abschn. 6.1.

2021-11-17:

- Erste Version mit dieser Historie.
- Der veraltete Abschn. 3.2 „Sichtbarkeit von Interaktor-Symbolen“ wurde durch einen Abschnitt ersetzt, der der Frage nachgeht, wann ein Interaktor besser an die Planungsgruppe oder an das Gruppenelement gebunden wird.
- Die Abschnitte 3.3 und 3.4 wurden getauscht.
- Aktualisierung des Abschnitts 3.3 zu den dynamischen Interaktoren, inklusive Hinweisen zur Verwendung des Platzhalters `$INTERACTOR`.
- Ergänzender Hinweis im Abschn. 4.4 zu Aktionen, die das aktive Objekt entfernen.
- Neuer Abschnitt 4.6 zur Verwendung des Dummy-Aktionstyps *NoAction*.
- Die Klasse *xOiTabularPlGroup* wird in Abschn. 5 nun „gleichberechtigt“ mit den älteren Klassen für Planungsgruppen behandelt.
- Ergänzender Hinweis im Abschn. 5.1 zur Verwendung der Objektkategorie *MethodCall*.
- Ergänzender Hinweis im Abschn. 5.5 zur Methode *xOiLayoutGroup::replaceElement()*.
- Umfangreicher Aus- und Umbau von Abschn. 5.9 mit vielen neuen Hinweisen und Tipps zur Anlage und Verwendung von Gruppenmerkmalen (Gemeinsame Eigenschaften).
- Umbenennung von Abschn. 5.11 (bessere Beschreibung des Inhalts) und Ergänzung eines Beispiels.
- Ergänzender Hinweis im Abschn. 5.12 zur Verwendung der Optionen *@ROPropsEditable4Common* und *@NonVisibleProps4Common*.
- Neuer Abschn. 5.13 mit Hinweisen zur Realisierung von Reaktionen auf Merkmalsänderungen von Elementen seitens einer Gruppeninstanz.