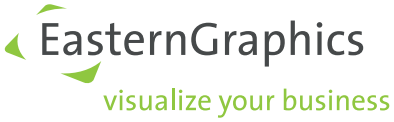

The OFML Interface *Property*

Version 2.9

Thomas Gerth, EasternGraphics GmbH (Editor)

2023-09-14

Contents

References

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

[dsr] Data Structure and Registration (DSR) Spezification. EasternGraphics GmbH

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

The documents are available at the Download Center of EasternGraphics

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

in the category *OFML Specifications*.

1 Introduction and General definitions

The specifications in this document replace section 4.4[1] in part III of the OFML specification [ofml]. However, the methods specified there still may be used. For the sake of completeness, they are listed in section B, but are no longer specified here. If necessary, consult [ofml].

This document version refers to version 1.43 of OFML base library OI (fall 2023).

1.1 Motivation to revise the interface

- Replacement of method *getProperties()* with its string-based interface by separate methods for calling up the property definitions and for calling up the choicelists, where each method returns a specific structure.
  (This eliminates the parsing of the string-based property definitions and, thus, avoids problems with special characters in values and texts.)
- Correspondingly, separate methods for the definition of the property and for the definition of the choice list of a property (if available).
- Correction of inconsistencies in the previous property definitions.
- Support for new requirements resulting from projects resp. further development of OCD:
  - multiple ranges
  - raster
  - representation of non-selectable values in choice lists
  - presentation of extra charges for values in choice lists
- New property type for choice lists whose values are Strings.
  (The need to convert OCD values into OFML symbols would then be omitted and associated problems would be avoided.)
- Improved handling of languages.

1.2 Attributes of properties

The attributes of properties can be subdivided as follows:

1. invariable attributes that determine the type of a property: type of values, formatting specifications, etc.
2. attributes that depend on the configuration of the object/article: (current) value, choice list, value range(s), state, position in the list of properties
3. language-dependent names for property and choice list values

In the interface, the attributes of the first group always are transferred in the form of a Vector, which is referred to as the *property definition* (see next section).

There are each separate methods for setting and recalling the property definition and the other attributes.

[1] which is referred to in this document as the *told* or *previous* property interface

1.3 The property definition

As explained in the previous section, the property definition summarizes all invariant attributes that determine the type of a property. These always are assigned and retrieved together by means of a Vector, which has the following structure:

```
[<type>, <width>, <dec-places>, <choice-list-type>]
```

The meaning of the attributes is as follows:

1. **type** (String) – type of the property:

   The property type defines the basic way of entering and displaying values as well as their OFML data type.

   The following types are defined (corresponding OFML data type in round brackets):

   L length in meter (Float)

   The value (in *m*) is formatted and displayed according to the user settings (in particular according to the specified unit). When entering, the value converted into *m* is rounded to the number of decimal places specified in `dec-places` before it is transferred to OFML[2]. Attribute `width` is ignored.

   A angle in radian (Float)

   The value (in *rad*) is formatted and displayed according to the user settings (in particular according to the specified unit). When entering, the value converted into *rad* is rounded to the number of decimal places specified in `dec-places` before it is transferred to OFML[2]. Attribute `width` is ignored.

   N number (Float, Int)

   Depending on `dec-places` it is an integer or a number with decimal places. Attribute `dec-places` defines the number of decimal places to be displayed. When entering, the entered value is rounded to the specified precision. If `dec-places` has the value `0`, the representation does not use thousand separators, and entering characters other than decimal digits and a sign in the first place is prevented. Apart from that, an entered value is displayed and parsed according to the rules of the current oprating system locale.
   Attribute `width` is ignored.

   B logical value (Int`{0,1}`)

   Attributes `width`, `dec-places` and `choice-list-type` are ignored[3]. The representation and the way of input are left to the applications.

   S character string (String)

   Attributes `dec-places` and `choice-list-type` are ignored[4]. The current property value always is displayed entirely, regardless of attribute `width`. Of the escape sequences possible in OFML Strings, only `\"` (double quote), `\'` (apostrophe) and `\\` (backslash) are allowed. If the value String contains other escape sequences (e.g. a newline escape sequence), the behavior is undefined.
   When entering characters, adding characters is prevented if `width` is reached or exceeded. In the case of Unicode, `width` is the count of code points after conversion to NFC.

   T (multiline) text (String)

   Attributes `width`, `dec-places` and `choice-list-type` are ignored[5]. A newline escape sequence in the current property value leads to a line break in the display. Conversely, when passing to OFML a line break is replaced by a newline escape sequence.

[2] Whether and in what form attribute `dec-places` also is considered for display is up to the applications.
[3] The corresponding elements of the property definition should have the values `1`, `0` resp. `@None`.
[4] The corresponding elements of the property definition should have the values `0` resp. `@None`.
[5] The corresponding elements of the property definition should have the values `0`, `0` resp. `@None`.

Y, YS symbolic choice list (Symbol, String)

Attributes `width` and `dec-places` are ignored[6].

`choice-list-type` has to be `@FixedSingleV` or `@FixedMultiV`.

`"Y"` and `"YS"` essentially behave identically. The difference is that in case of `"Y"` the value has type *Symbol*, while in case of `"YS"` it has type *String*.

CT ... special (custom) type (Any)

This type can be used to define non-standardized or proprietary property types. `"CT"` is followed by information on the special type. Attributes `width`, `dec-places` and `choice-list-type` also may be used.

The availability of a special type in the currently used application can/should be checked on the OFML side using application callback *::ofml::app::isPropTypeSupported()* (see section 3.4).

2. **width** (Int) – maximum input length:

   Relevant only for properties of type `"S"`: defines the maximum input length of the character string[7]. For all other types, a more or less reasonable value has to be specified for this attribute in the property definition[8].

3. **dec-places** (Int) – number of decimal places:

   Relevant only for properties of types `"L"`, `"A"` and `"N"`[9].

4. **choice-list-type** (Symbol) – type of choice list:

   Determines whether a choice list exists and, if so, of what type it is.
   Is not relevant for properties of types `"B"`, `"S"` and `"T"`[10].

   @None No choice list available.

   Not valid for properties of types `"Y"` and `"YS"`.

   @FixedSingleV Fixed choice list, only a single value can be selected.

   The input of additional values is not allowed.

   @FixedMultiV Fixed choice list, multiple values can be selected.

   The input of additional values is not allowed.

   Valid only for properties of types `"Y"` snd `"YS"`.

   @OpenSingleV Open choice list, only a single value can be selected.

   The input of additional values is permitted (within the specified value ranges).

   Not valid for properties of types `"Y"` and `"YS"`.

   The definition of a property with an invalid combination of property type and choice list type is rejected, i.e. the call of method *setupProperty2()* (see 2.1) then has no effect.

[6] `width` should have a value that is not smaller than the length of the longest of the possible symbolic values. The value for `dec-places` should be `0`.
[7] However, using *setPropValue()* a character string can be assigned to the property that is longer than `width`.
[8] If no reasonable value can be determined, then value `0` should be used.
[9] For all other types, value `0` has to be specified in the property definition.
[10] For these types, value `@None` has to be specified in the property definition.

The following table shows the possible combinations of the attributes with the respective old property type.

type	width	dec-places	choice-list-type	old property type	OFML date type
L	-	-	@None	"f", Fmt "@L"	Float
L	-	-	@FixedSingleV	"u chf", Fmt "@L"	Float
L	-	-	@OpenSingleV	"u chf.edit", Fmt "@L"	Float
A	-	-	@None	"f", Fmt "@A"	Float
A	-	-	@FixedSingleV	"u chf", Fmt "@A"	Float
A	-	-	@OpenSingleV	"u chf.edit", Fmt "@A"	Float
N	-	0	@None	"i"	Int
N	-	0	@FixedSingleV	"u chi"	Int
N	-	0	@OpenSingleV	"u chi.edit"	Int
N	-	n	@None	"f"	Float
N	-	n	@FixedSingleV	"u chf"	Float
N	-	n	@OpenSingleV	"u chf.edit"	Float
B	-	-	-	"b"	Int
S	n	-	-	"s"	String
T	-	-	-	"t"	String
Y	-	-	@FixedSingleV	"ch", "chf"	Symbol
Y	-	-	@FixedMultiV	"mch"	Symbol[]
YS	-	-	@FixedSingleV	("ch", "chf")[a]	String
YS	-	-	@FixedMultiV	-	String[]
CT	*	*	*	"u ..."	*

[a] Although the (old) OFML specification actually only allows Symbols as choice list values, the applications of EasternGraphics also support choice lists whose values are Strings (without language-specific value descriptions). This is used, for example, in Metatype-based data creation. According to the new interface, such properties behave in principle like properties of the type "YS", with the value and language-specific value description being identical.

1.4 Language–Text mappings

Language-specific names of properties and property values either are realized using a text resource ID or refer to the language that currently is to be used for product data texts according to method *OiPlanning::getPDLanguage()*, see 3.4.

However, some clients require texts in all available languages in order to work efficiently. For this, the return structures of some methods in section 2.7 contain so-called *language–text mappings*.

A language–text mapping consists of a *Vector* of zero or more elements, where each element again is a *Vector* made up of two elements:

1. language code (*String*) according to ISO 639-1 Alpha-2 or an empty String (see below)
2. text in the corresponding language (*String*)

The language code, if not empty, has to consist of two lowercase letters. Combinations of letters that do not correspond to an officially registered code are not permitted, but should not lead to errors when processed by the client.

An empty string in the 1st element is synonymous with an *undefined language*. The associated texts can be used by the client as a fallback in the case that the mapping does not contain any text for the (or one) language required by the client.

A language–text mapping must not contain two entries with the same language code!

2 The methods

The methods that were not yet included in the previous interface specification are marked with **new**.

Some methods have been renamed in order to standardize the naming. This affects, amongst others, the use of "Property" in the method name: If this word is followed by another word (e.g. "State"), it is abbreviated as "Prop".

If the purpose is the same but the specification is different, the name of the new method is derived from the name of the previous method by adding "2".

2.1 Property setup

The methods in this group are used to set up a property including position and activation state as well as to call up the corresponding attributes. Further methods for convenient, simultaneous call up of several attributes can be found in method group "client support" (see section 2.7).

- *setupProperty2(pKey(Symbol), pDef(Any[4]), pName(String), pPos(Int), pState(Int)) → Void* **new**

  The method creates a property with the specified key (identifier) and assigns it the given definition, language-specific name, position and activation state.

  If there is already a property registered with the specified key but a different definition, the method call has no effect.

  The definition of a property (parameter *pDef*) is a Vector made up of four elements. The structure of the vector and the meaning of the elements is described in detail in section 1.3.

  The language-specific name of the property (parameter *pName*) can be changed later with separate method *setPropName()*. For details on using the parameter, see there (below).

  The position of the property in the property list (parameter *pPos*) can be changed later with separate method *setPropPos()*. For details on using the parameter, see there (below).

  The activation state of the property (parameter *pState*) can be changed later with separate method *setPropState2()*. For details on using the parameter, see section 2.2.

  The language-specific name, the position and the state also can be changed later using separate methods (see below).

- *getPropDef2(pKey(Symbol) → Any[4] | Void* **new**

  The method returns the definition of the property with the specified key. The structure of the returned Vector corresponds to the structure of parameter *pDef* passed as the property definition to method *setupProperty2()*.

  If no property with the specified key is defined for the implicit instance, the return value has type *Void*.

- *removeProperty(pKey(Symbol)) → Void*

  The method removes the property with the specified key from the property list.

  If no property with the specified key is defined for the implicit instance, the method has no effect.

- *clearProperties() → Void*

  The method removes all properties from the property list.

- *getPropKeys() → Symbol[]* **new** [11]

  The method returns a List containing the keys of all properties currently defined for the implicit instance.

[11] is identical to previous method *getPropertyKeys()*

The properties are sorted in ascending order according to their explicit positions. Properties without an explicit position appear at the end of the list in an undefined order.

- *hasProperties() → Int*

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

- *hasProperty(pKey(Symbol)) → Int*

  The method returns `1`, if a property with specified key is defined for the implicit instance, otherwise it returns `0`.

- *setPropName(pKey(Symbol), pName(String)) → Void* **new**

  The method assigns a language-specific name to the property with the specified key.

  Parameter *pName* can be a text resource ID, which then is resolved language-specifically by the OFML runtime environment using external resource files. If the parameter is not a text resource ID, the name has to be given in the language that currently is to be used for product data texts according to function *getPDLanguage()* of interface *Article* [article] (see also 3.4).

  If no property with the specified key is defined for the implicit instance, the method has no effect.

- *getPropName(pKey(Symbol)) → String | Void* **new**

  The method returns the language-specific name, which currently is assigned to the implicit instance for the property with the specified key.

  If no property with the specified key is defined for the implicit instance, the return value has type *Void*.

- *setPropPos(pKey(Symbol), pPos(Int |Void)) → Int | Void* **new** [12]

  The method specifies the desired position in the property list for the property with the given key.

  If no property with the specified key is defined for the implicit instance, the method has no effect and return value has type *Void*.

  If a property with the specified key is defined for the implicit instance, the old position information is overwritten. If parameter *pPos* is an integer greater than or equal to `0` and if the desired position already has been assigned for another property, then that and all subsequent properties in the position list are moved back by one position. If *pPos* has type *Void* or if the parameter has the value `-1`, no special position is required for the property. Then it is sorted in the property list after the properties for that a position explicitly was requested.
  Return value is the new position of the property or `-1` if no special position is required.

- *getPropPos(pKey(Symbol)) → Int | Void* **new**[13]

  The method returns the position of the property with the specified key.

  If no special position was requested for the property via *setupProperty2()* or *setPropPos()*, the return value is `-1`.

  If no property with the specified key is defined for the implicit instance, the return value has type *Void*.

- *setExtPropOffset(pOffset(Int)) → Void*

  This method assigns an offset for positions of externally defined properties, i.e. properties that are defined by another instance for the implicit instance.

  The offset specifies the smallest position number that may be used for externally defined properties.

  The default offset is `0`.

[12] is identical to previous method *setPropPosOnly()*
[13] is identical to previous method *getPropertyPos()*

- *getExtPropOffset() → Int*

  The method returns the offset for positions of externally defined properties, i.e. properties that are defined by another instance for the implicit instance.

  The offset specifies the smallest position number that may be used for externally defined properties. This offset must be called up from an external instance before a property is defined for the implicit instance and must be taken into account when assigning explicit positions.

2.2 Activation state

- *setPropState2(pKey(Symbol), pState(Int)) → Void* **new**

  The method assigns the given activation state to the property with the specified key.

  If no property with the specified key is defined for the implicit instance, the method has no effect.

  The state is a combination (addition) of the following flags (bits):

  `1` the property is visible

  `2` the property can be edited by the user

  `4` the input of a value still is required so that the configuration is complete or valid

  `8` the property currently is not defined for the implicit instance (but can be present in another configuration)

  The flag with the value `8` is reserved for the return structures of certain methods in section 2.7. In parameter *pState* of *setPropState2()* this flag always must have the value `0`!
  Therefore, the possible combinations for *setPropState2()* are:

visible	editable	input required	property not def.	state value (old)	description
0	0	0	0	0 (-1)	for internal use only
1	0	0	0	1 (0)	visible, but not editable
1	1	0	0	3 (1)	visible and editable
1	1	1	0	7 (n.a.)	visible and editable, input still required

  If an invalid value is passed in parameter *pState*, the method has no effect.

- *getPropState2(pKey(Symbol)) → Int* **new**

  The method returns the current activation state of the property with the specified key.

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

- *invalidateProperties() → Void*

  The method sets the activation state of all currently defined properties of the implicit instance whose current state is `3` (active/visible) to `1` (inactive/visible).

  This method can be used by clients to prevent the configuration of article instances that could not be updated.

2.3 Value ranges and choice lists

- *setPropRanges(pKey(Symbol), pRanges(Any))* → *Void* **new**

  The method assigns zero, one or more[14] value ranges to the numeric property with the specified key.

  The relevant property types are `"L"`, `"A"` and `"N"`. The choice list type of the property must not be `@FixedSingleV`[15].

  If the choice list type of the property is `@OpenSingleV`, the choice list may contain values that are outside the assigned value ranges[16].

  Parameter *pRanges* either is of type *Void* or a Vector, consisting of one or more Vectors with 3 elements, each defining a value range:

  1. minimum
  2. maximum
  3. increment (raster)

  The specified values for minimum and maximum belong to the range of valid values, i.e., comparison operators `>=` resp. `<=` are applied.

  One of the two values for minimum or maximum can be of type *Void* to indicate a range that has no lower resp. upper limit.

  If the 3rd element is of type *Void*, there are no further restrictions with regard to the permitted values within the specified value range. If a minimum is specified (not of type *Void*), an increment can be specified in the 3rd element that, starting from the minimum, must be adhered to when entering values, thus limiting the permitted values within the specified range[17].

  Invalid value range definitions in parameter *pRanges* are ignored. Specifically, this applies to the following situations:

  – Both minimum and maximum are of type *Void*.
  – The minimum is of type *Void* and the increment is not of type *Void*.

- *getPropRanges(pKey(Symbol))* → *Any* **new**

  The method returns the value ranges currently defined for the numeric property with the specified key.

  The return value is of type *Void* if the implicit instance has no property with the specified key, or if no value ranges have been assigned to the property using method *setPropRanges()*. Otherwise the return value corresponds to parameter *pRanges* of method *setPropRanges()*.

- *setPropChoiceList(pKey(Symbol), pChoiceList(Any))* → *Void* **new**

  The method assigns a choice list to the property with the specified key.

  The method only has an effect if the current choice list type of the property is *not* `@None`!

  The choice list can be passed explicitly as a list of *value specifications* (see below) or be specified indirectly by passing a method of the implicit instance that returns the list of value specifications. In the latter case parameter *pChoiceList* has type *String* and contains the name of the method including parameter brackets and possible constant parameter values[18]. Per default, the method is called (every time) when *getPropChoiceList()* is called (see below)[19].

[14] The property editors of the applications of EasternGraphics currently support only a single value range.

[15] If the choice list type is `@FixedSingleV`, this method has no effect.

[16] because in this case the value ranges define the areas in that the user freely may enter values *in addition* to the values from the choice list.

[17] The property editors of the applications of EasternGraphics currently do not support a raster in a value range. If necessary, a choice list should be specified in that the raster values explicitly are included.

[18] OFML programmers should note that the choice list method now has to return a list with value specifications, *not* a String with the choice list representation according to the old property interface (method *setupProperty ()*).

[19] In contrast to methods *getProperties()* and *getPropertyDef()* of the old interface, the call of the method is not left to the client, but already is executed within *getPropChoiceList()*. This is necessary on the one hand for convenience method *getPropChoiceList2()* (see section 2.7), on the other hand also in order to support the old property definition.

A **value specification** is a Vector consisting of the following elements:

1. value (*Any*)
2. language-specific description (*Void* | *String*)
3. state (*Int*)
4. amount of extra charge (*Void* | *Float*)
5. currency of extra charge (*Void* | *String*)

The type of the value (1st element) results from the property type of the current property definition. However, the value also can be of type *Void* if the user is to be allowed to explicitly set the unevaluated state of the property[20].

The language-specific description (2nd element) is optional. If a language-specific description is given (type *String*), it is displayed in the property editor instead of the value itself, with the following restrictions:

- With property types `"L"` and `"A"` any given language-specific description is ignored.
- With choice list type `@OpenSingleV` (property type `"N"`) a possibly specified language-specific description is displayed only in the unfolded choice list itself.

The description should be given in the language that currently is to be used for product data texts according to function *getPDLanguage()* of interface *Article* [article] (see also 3.4). If no language-specific description is given (type *Void*), with property types `"Y"` and `"YS"` the value is interpreted as a text resource ID, which is resolved language-specifically by the OFML runtime environment using external resource files. If the text resource could not be resolved or if the property type is not `"Y"` or `"YS"`, the value itself is displayed in the choice list of the property editor.

The *state* in the 3rd element is a combination (addition) of the following flags (bits):

`1` the value is selectable

If the flag has the value `0`, the choice list value should be displayed in the choice list, but it must not be selectable. In the property editors, these values are to be set apart in a suitable manner from the values that actually can be selected, e.g. by a grayed-out display.

`2` the value is not contained in the current choice list of the implicit instance (but can be contained in another configuration)

This flag is reserved for the return structures of certain methods from section 2.7. When passed to *setPropChoiceList()*, this flag always must have value `0`!

Thus, the possible values for the 3rd element when passed to *setPropChoiceList()* are `0` and `1`. (An entry in the choice list with a different state value is ignored.)

Elements 4 and 5 are optional. If they are not of type *Void* they specify the amount and currency of the surcharge, which is claimed when the value is selected. The property editors must display this surcharge in the choice list in a suitable way (e.g. enclosed in brackets after the value or its description). If necessary, a conversion into the currency that currently is set in the application has to be executed.

- *getPropChoiceList(pKey(Symbol), ...)* → *Any* **new**

The method returns the choice list currently defined for the property with the specified key.

The return value is of type *Void* if the implicit instance has no property with the specified key or if no choice list has been assigned to the property using method *setPropChoiceList()* .

If a choice list explicitly was passed the last time *setPropChoiceList()* was called, this list directly will be returned.

Otherwise, if the choicelist is defined indirectly by specifying the name of a method, the behavior depends on the optional parameter (of type *Int*):

- If no optional parameter is specified (default) or if it has the value 1, this method is called first and then its result is returned.
- If the optional parameter has the value 0, the name of the method is returned[21].

[20] This is used, for example, for numerical restrictable OCD properties that currently are not evaluated, see section 3.2.

[21] i.e., the return value is the same as the value passed in parameter *pChoiceList* at last call to *setPropChoiceList()*

2.4 Property values

- *getPropValue2(pKey(Symbol)) → Any* **new**

  The method returns the value currently stored in the implicit instance for the property with the specified key.

  The return value can be of type *Void* if currently no regular value is assigned to the property according to its type resp. its value range[22].

  If no property with the specified key is defined for the implicit instance, the return value is of type *Void*.

  If the type of the implicit instance has a get-method that matches the property

  get<Key>() → Any

  this method is used by the standard implementation in order to retrieve the current value.

  If the type of the implicit instance does not have such a method, the value is retrieved from the Hash table for dynamic properties (see method *getDynamicProps()* in interface *Base*).

  See also method *forceDynamicProp()* in section 2.6.

- *setPropValue(pKey(Symbol), pValue(Any)) → Int*

  The method assigns a new value to the implicit instance for the property with the specified key.

  The standard implementation of this method executes the following actions:

  1. Verification of compliance with the maximum input length (attribute `width`) for properties of type `"S"` as well as verification of compliance with value ranges and increment (raster) for numeric properties (`"L"`, `"A"`, `"N"`). If a violation of the restrictions is detected, the method is terminated without further actions.
  2. If the value to be assigned is equal to the current value of the property, the method is terminated without further actions.
  3. For properties with a choice list of type `@FixedSingleV` or `@FixedMultiV`, which was explicitly assigned as a list of value specifications (see 2.3), it is checked whether the value to be assigned is contained in the choice list.
     If this is not the case, the method is terminated without further actions.
  4. Actual value assignment.
     If the type of the implicit instance has a set-method that matches the property
     set<Key>(pValue(Any)) → Void
     this method is used in order to assign the value. As a rule, this method only assigns the value to a corresponding instance variable. Any further semantics, such as the re-creation of the geometry or collision tests, are reserved for method *propsChanged()* (see below).
     If the type of the implicit instance does not have a suitable set-method, the value is written into the Hash table for dynamic properties using the key of the property as the hash key.
     See also method *forceDynamicProp()* in section 2.6.
  5. If the property is associated with a property in the product data (OCD), the global product data manager then evaluates relationships between properties (as a result of which other properties resp. their values can change).
  6. Then method *propsChanged()* (see below) is called to carry out special treatments. For parameter *pDoChecks* value `1` (True) is passed.
  7. If the value assignment has been rejected by the product data manager or by method *propsChanged()*, all properties are reset to the state at the beginning of *setPropValue()* and then method *propsChanged()* is called again, but this time value `0` (False) is passed for parameter *pDoChecks*.

  Return value of the method is `1` if the definition of one or more properties has changed or if properties have been added or removed. Otherwise the return value is `0`.

[22] In this case, the standard implementation of obsolete method *getPropValue()* returns the special value `@VOID`, which represents a type conflict with non-symbolic properties. For reasons of backward compatibility, this implementation cannot be changed, what is the reason why new method *getPropValue2()* is introduced.

Some basic OFML types realize the following additional behavior:

Interface Article (OiPart, OiPlElement)
: If the update state of the article instance is *@Undefined*, before the actual value assignment first an attempt is made to update the article instance. The standard actions (see above) only take place if the update state of the article instance in the result of that attempt is *@Up2Date*.

OiPart, OiPlElement, OiProgInfo, OiPropertyObj
: If there are properties whose values have changed after the standard actions (see above), an event of type *@PropertyChange* is triggered by the *OiChangeManager*, passing the respective instance as the publisher and the list of changed properties as the event argument.

OiPlElement
: If there are properties whose values have changed after the standard actions (see above), any existing dimensions are updated (adaptation to possibly changed dimensions).

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

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

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

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

  The standard implementation performs the following checks:
  - Does the type of the value match the type of the property?
  - With properties of type `"S"`:
    Is the maximum input length (attribute `width`) observed?
  - With properties with a choice list of type `@FixedSingleV`:
    Is the passed value included in the choice list?
  - With (numeric) properties with value ranges:
    Is the passed value included in the defined value ranges and does it match any specified increment (raster)?

- *propsChanged(pPKeys(Symbol[]), pDoChecks(Int)) → Int*

  The method carries out special treatments and checks after changing property values. The properties whose values have been changed are specified by their keys. Parameter *pDoChecks* specifies whether checks have to be carried out or whether it is only necessary to react to changes, e.g. by adjusting the geometry.
  Return value is `1` if the new property values are valid. Otherwise value `0` has to be returned.

  > Note:
  > The method is called from the standard implementation of method *setPropValue ()* (see above). Method *propsChanged()* usually adjusts the geometry or the material characteristics of the implicit instance.

- *changedPropList() → Symbol[]*

  The method returns a reference to the List of properties whose values have changed during the processing of method *setPropValue()*. The properties are specified in the list using their keys.

  > Note:
  > Actually, the method is used only by the product data manager during the evaluation of product data relationships within method *setPropValue()*.
  >
  > The list is cleared at the beginning of each execution of *setPropValue()*.

2.5 Property classes and groups

A *property class* is a set of properties that are grouped together from a logical, conceptual or other point of view.
The concept of class is technical. (For example, a class does not have a language-specific description.)

Property groups are sets of properties that are grouped together with regard to the user, and which are (can be) used for grouping by the property editor of an OFML application.

In contrast to the static character of the classes, the division of the properties into groups can change dynamically, e.g. depending on the current configuration or on the currently set language.

A property can be assigned to exactly one class and one group. (However, a property does not have to be assigned to a class or group.)
The assignment to a class is done using the method *setPropClass()*, to a group using the method *getPropGroupDescriptions()* (both see below).

- *setPropClass(pPKey(Symbol), pPClass(String))* → *Void*

  The method assigns the property with the given key to the specified class.

- *delPropClass(pPKey(Symbol))* → *Void*

  The method removes the property with the given key from the specified class. (The method has no effect if the specified property currently is not assigned to the specified class.)

- *getPropClass(pPKey(Symbol))* → *String* | *Void*

  The method returns the property class to which the property with the specified key currently is assigned, or a value of type *Void* if the property currently is not assigned to a class.

- *getPropClasses()* → *String[]*

  The method returns a *List* of all property classes to which the properties of the implicit instance currently are assigned.

- *getPropGroupDescriptions(pLanguage(String))* → *Any[]*

  The method returns a *List* with descriptions for the currently defined property groups of the implicit instance.

  Each element of the list describes a property group and is a *Vector* consisting of the following elements:

  1. name (identifier) of the group (*String*)
  2. language-specific (short) description of the group (*String*)
  3. list of the keys of the (currently defined) properties that belong to the group (*Symbol[]*)

  The order of the groups in the list defines the order in which the groups should be displayed in the property editor of an OFML application.
  In turn, the order of the properties in element 3 of a group defines the order in which the properties should be displayed within the group.

  Properties that are not explicitly assigned to a specific group are assigned to the dummy group `OI_NONE_PROPCLASS`[23]. The language-specific description (2nd element) for this group is the same as the identifier of the group. The clients (applications) must use their own text resources for this group.

[23] The name of this group better should be `OI_NONE_PROPGROUP`. However, for reasons of backward compatibility renaming of the previously used group is/was not done.

The *standard implementation* of the method uses the following procedure:

a) The return of specific groups is prevented if method *needPropGroupDescriptions()* of the *ProgInfo* object responsible for the implicit instance returns value `0` for this instance. In this case, all (currently defined) properties of the implicit instance are assigned to dummy group `OI_NONE_PROPCLASS`.
The positions of the properties within this group result from the positions in the property list assigned to them, see method *setPropPos()* (section 2.1).

b) To determine the groups, first a request is made to the global product data manager.
The result may also contain properties that are not assigned to any explicit group. These are assigned to the group `OI_NONE_PROPCLASS`.

c) For the properties that were not handled by the product data manager, the groups are determined as follows:

- Properties that are not assigned to a class are assigned to the dummy group `OI_NONE_PROPCLASS`.
  Otherwise, a group with the same name is defined for the class of the property[24] and the property is assigned to this group.
- If possible, the language-specific description (2nd element) for a group derived from a class is delivered in the language that is specified in the parameter of the method. It is determined as follows:
  1. Calling a method on the product database of the series of the implicit instance to query a language-specific description for the class.
     If this method returns a value of type *Void*[25]:
  2. Search for a text resource in the namespace of the implicit instance, using the class identifier as the resource key.
     If no text resource was found:
  3. Use of the identifier of the class itself.

  If the method of the product database returns an empty string (step 1), all properties of the class are assigned to group `OI_NONE_PROPCLASS` (and the class/group is removed from the return list).

d) Groups that were defined by the product data manager follow the groups that were derived from classes (see above) and the group `OI_NONE_PROPCLASS`. In doing so, the order defined by the product data manager is retained.

The position in the return list of a group not defined by the product data manager (incl. `OI_NONE_PROPCLASS`) results from the lowest number of the positions of the properties contained each, i.e., the group that contains the property with the smallest position number comes first.

d) The position of a property in a group explicitly defined by the product data manager is predefined by the product data manager.

The positions of properties in a group not defined by the product data manager (incl. `OI_NONE_PROPCLASS`) result from the positions in the property list assigned to them, see method *setPropPos()* (section 2.1).

f) Finally, if the *ProgInfo* object responsible for the implicit instance implements method *getPropGroupDescriptions4Obj()*, this method is called passing the reference to the implicit instance, the requested language and the return list determined by the standard implementation up to this point. This can be used to implement series-specific behavior.

[24] if this group is not already defined
[25] for classes that are not defined in the product data

2.6 Miscellaneous methods

- *getPropInfo(pKey(Symbol), pPropValue(Any), pInfoType(Symbol))* → *Any*

  The method returns the information of requested type for the specified property value.

  The return value is of type *Void* if the instance does not possess the specified property or if no information of the requested type is available.

  The following standard information types are predefined:

  @Picture
  : Name of the image file that illustrates the property value (*String*)

  @Text
  : textual description (*String*, can be a text resource)

  @HTML
  : URL of the HTML description (*String*)

  Information types `@Text` and `@HTML` currently are not used.

  Format, image size and storage location of the image file (information type `@Picture`) are determined by the respective OFML application[26].

  The standard implementation delegates the call to the method *getPropInfo4Obj()* of the *OiProgInfo* instance that is responsible for the implicit instance (if available).
  The implementation of *OiProgInfo::getPropInfo4Obj()* uses corresponding entries in the control data table `proginfo.csv` for information type `@Picture`.

- *updateProperties()* → *Int*

  The method updates the properties of the implicit instance.

  The method is called by the property editor of the application *before* the properties of the implicit instance are displayed. Therefore, it offers the opportunity to react both to changes in the data and to changes in the runtime environment (e.g. changed language setting).

  The return value has no specific meaning.

  The standard implementation does not perform any actions and returns `1`.

  Base class *OiPlElement* implements the method as follows:
  If the language to be used for product data texts (method *OiPlanning::getPDLanguage()*, section 3.4) has been changed since the creation of the implicit instance or since the last call of *updateProperties()*, and if the update status of the implicit instance is `@Up2Date`, the global product data manager is called to update the commercial properties[27].
  The return value is `1`.

- *getPropTitle()* → *String*

  The method provides a brief description of the instance for use in the header of the property editor[28].

  The standard implementation returns an empty string.

  The implementation in the base class *OiPlElement* (interface *Article*) uses corresponding entries in the control data table `plelement.csv`, which control the generation of the headline. By default, the generated string is made up of the base article number and the article short text.

- *setPropHintText(pKey(Symbol), pHint(String | Void))* → *Void* **new**

  The method assigns a hint text to the implicit instance for the property with the specified key. The text can be displayed as a hint by an application if the user moves the mouse pointer over the name of the property in the property editor.

[26] For the applications of EasternGraphics this is uniformly regulated by the DSR specification [dsr].
[27] This is done in order to adapt the language-specific descriptions for properties and values to the new/current language.
[28] It is up to the applications to decide whether the method is actually used. They can use a different scheme when designing the header.

The text can consist of several lines (i.e. contain newline escape sequences).

The text has to be given in the language that currently is to be used for product data texts according to function *getPDLanguage()* of interface *Article* [article] (see also 3.4).

If a value of type *Void* is passed in parameter *pHint*, any previously assigned hint text is removed.

- *getPropHintText(pKey(Symbol))* → *String* | *Void* **new**

  The method returns the hint text that currently is assigned to the implicit instance for the property with the specified key (see *setPropHintText()* above).

  The return value has type *Void* if no property with the specified key is defined for the implicit instance, or if currently there ist no hint text assigned for this property.

- *forceDynamicProp(pPKey)* → *Int*

  The method returns `1` (true) if the value of the property with the specified key is to be stored in the hash table for dynamic properties[29]. In this case, possibly existing set- resp. get-methods are ignored.

  By this means, the false use of set- resp. get-methods can be prevented, that serve a purpose other than storing a property value in a member variable.

  > This is of particular importance for classes that are used to represent articles (interface *Article*), since the global product data manager dynamically can define properties that are associated with commercial characteristics. As the programming of the class and the creation of the commercial data usually are carried out separately, it can happen that a commercial characteristic is created with a name for which there are already suitable set- resp. get-methods, but which serve a completely different purpose. Therefore, it is strongly recommended to override this method in all classes that implement interface *Article* and that define new set- resp. get-methods (which are not used to store a property value).

  If the method is overwritten, the inherited implementation has to be called for all property keys that are not relevant for the class!

- *resetProperties()* → *Void*

  The method (re)sets the state (attributes, values) of all properties of the implicit instance to the initial state (default).

  The definition of the default state is type-specific.

  The standard implementation takes no action.

  The implementation in base OFML classes *OiPlElement* and *OiPart* (interface *Article*) first delegates the request to the global product data manager (creation of the initial state of the article) and then to the global planning instance (assignment of application resp. user-specific defaults, e.g. based on configuration profiles).
  The term *default* refers to the article that currently is represented by the implicit instance, not to the article that was represented by the instance when it was created[30].

[29] see method *getDynamicProps()* in interface *Base*
[30] With *article-polymorphic* classes, the article number of an instance can change during its lifetime.

2.7 Client support

The methods of this group combine two or more methods of the previous groups and, thus, provide a more effective way to query property attributes.

- *getPropSpec(pKey(Symbol))* → *Any* **new**

  The method returns the extended specification of the property with the specified key.

  In addition to the actual property definition, the extended specification – amongst others – also includes the activation state, the position, value ranges, the current value (along with an optional value description) and the (optional) name of the image file (for the current value).
  (The methods whose functionality is included in *getPropSpec()* are mentioned below for the individual elements of the return structure.)

  If no property with the specified key is defined for the implicit instance, the return value is of type *Void*, otherwise it is a Vector currently consisting of the following elements[31]:

  1. property type (*String*)
  2. maximum input length (*Int*)
  3. number of decimal places (*Int*)
  4. type of choice list (*Symbol*)
  5. language-specific name (*String*)
  6. position (*Int*)
  7. activation state (*Int*)
  8. value ranges (*Any*)
  9. current value (*Any*)
  10. language-specific description for the current value (*Void* | *String*)
  11. name of the image file that illustrates the current value (*Void* | *String*)
  12. language-specific hint text for the property (*Void* | *String*)
  13. property class (*Void* | *String*)

  Elements 1 to 4 correspond to the actual property definition, which is returned by method call `getPropDef2(pKey)` (see section 1.3).

  Elements 5 and 6 correspond to the result of method calls `getPropName(pKey)` resp. `getPropPos(pKey)` (see 2.1).

  Element 7 corresponds to the result of method call `getPropState2(pKey)` (see 2.2).

  Element 8 corresponds to the result of method call `getPropRanges(pKey)` (see 2.3).

  Element 9 corresponds to the result of method call `getPropValue2(pKey)` (see 2.4).

  If it is not of type *Void*, element 10 contains the language-specific description, which is to be displayed instead of the current value itself.
  If the current property value (element 9) is of type *Void*, element 10 must *not* necessarily also be of the type *Void*, but can contain a string that describes the current state[32].

  For properties with a choice list of type `@FixedSingleV` the description is determined using the following procedure[33].

  1. If a language-specific description (non-empty string) is specified in the choice list for the value, this is used, otherwise:
  2. Call of (obsolete) method *getChLPropValueText()* (appendix B).
     If this call returns a value of the type *String* and the string is not empty, this is used, otherwise:

[31] If needed, the vector can contain additional elements in the future.
[32] This is used, e.g., for currently unassigned numerical restrictable OCD characteristics, see section 3.2.
[33] In applications that still use the old interface, this was done in the property editor itself. This is no longer necessary in applications that use the new interface, as this is encapsulated by the method *getPropSpec()*.

3. Determination of the text resource for the Symbol or the String (insofar as the passed value is of these two types).
   If a text resource could be determined, this is used, otherwise:
4. The string representation of the value is used.

   In steps 1 and 2, the language is used that is returnd by method *OiPlanning::getPDLanguage()* (see 3.4).

Element 11 corresponds to the result of method call `getPropInfo(pKey, <element 9>, @Picture)` (see 2.6).

Element 12 essentially corresponds to the result of method call `getPropHintText(pKey)` (see 2.6). If necessary, an additional text can be generated that describes, e.g., the currently defined value ranges of a numerical property.

Element 13 corresponds to the result of method call `getPropClass(pKey)` (see 2.5).

- *getPropSpecs()* → *Any[]* **new**

  The method returns a List with the extended specifications of the properties currently defined for the implicit instance.

  The elements in the list each describe a property and consist of a two-place Vector:
  1. property key (*Symbol*)
  2. property specification (*Any*), corresponds to the result of method call `getPropSpec(<element 1>)` (see above)

  The properties are sorted in the list according to their positions (i.e., the order corresponds to the order of the list returned by *getPropKeys()*, section 2.1).

- *getPropChoiceList2(pKey(Symbol))* → *Any* **new**

  The method returns detailed information about the choice list currently defined for the property with the specified key.

  The method combines the functionality of *getPropChoiceList()* (see 2.3) and *getPropInfo()* for information type *@Picture* (see 2.6). (This saves the client separate calls of *getPropInfo()* for each individual value in the choice list).

  The return value is of type *Void* if the implicit instance has no property with the specified key, or if no choice list has been assigned using method *setPropChoiceList()*.
  Otherwise the return value is a List of Vectors[34]. Each vector contains the information about a value from the choice list and currently consists of the following elements[35]:
  1. value (*Any*)
  2. language-specific description (*Void* | *String*)
  3. state (*Int*: 0 = invalid, 1 = valid)
  4. amount of extra charge (*Void* | *Float*)
  5. currency of extra charge (*Void* | *String*)
  6. name of the image file that illustrates the value (*Void* | *String*)

  The meaning of elements 1 to 5 is described in more detail in the specification of method *setPropChoiceList()* (see 2.3).

  If no language-specific description (element 2) was given when specifying a value via *setPropChoiceList()* (element is of type *Void*), and if the type of the property is `"Y"` resp. `"YS"`, the value is interpreted as a text resource ID, which is resolved now by *getPropChoiceList2()* using external resource files. In doing so, the language is used that is returned by method *OiPlanning::getPDLanguage()* (see 3.4). If the resolution of the text resource is not successful, the value itself is used as its description (for which purpose the value is converted into a String with properties of type `"Y"`).

  Element 6 corresponds to the result of method call `getPropInfo(pKey, <element 1>, @Picture)` (see 2.6).

[34] Note: the list may be empty (see 3.3)!
[35] If needed, the vector can contain additional elements in the future.

- *getAllPropSpecs(pRequiredState(Int)) → Any[]* **new**

  The method returns a List with the extended specifications of all possible properties for the implicit instance (i.e., including those properties that currently are not defined for the implicit instance).

  Currently, the method considers only properties that are generated by the global product data manager for commercial characteristics (OCD) of the article, which is represented by the implicit instance.

  The return value corresponds to that of *getPropSpecs()* with the following changes/extensions:

  – Properties that are known to never have the state (see *getPropState2()*, section 2.2) required in parameter *pRequiredState* are not included in the return list.

  – Element 5 of a property specification (language-specific name) contains a language–text mapping (see section 1.4).

  – For properties that currently are not defined for the implicit instance, the flag (bit) with the value 8 is set in the activation state (see *setPropState2()*, section 2.2).
  Then, elements 9 to 11 have a value of type *Void*.

  – Element 10 of a property specification (language-specific description of the current value) either is a value of type *Void* or a language–text mapping.

  – Element 12 of a property specification (language-specific hint text) is a language–text mapping resp. an empty Vector if no hint texts are available.

- *getDefaultPropGroupSpecs() → Any[]* **new**

  The method returns a List with the descriptions of the standard groups, i.e. the groups that are defined independently of the configuration of the implicit instance and independently of a specific language[36].

  Each element in the return List is a Vector consisting of the following elements:

  1. identifier of the group (*String*)
  2. List of keys of the properties belonging to the group (*Symbol[]*)
  3. language–text mapping (*Any[]*) with the language-specific (short) descriptions for the group

  Currently, the method considers only properties that are generated by the global product data manager for commercial characteristics (OCD) of the article, which is represented by the implicit instance.

  All properties that are returned by method *getAllPropSpecs()* (see above) should also be contained in the groups that are returned by *getDefaultPropGroupSpecs()*.

- *getAllPropValueInfos(pKey(Symbol), pPClass(String)) → Any* **new**

  The method returns information about all possible values of the choice list for the property with the specified key and the specified class.
  (I.e., the method also provides information about the values that are not contained in the current choice list of the implicit instance).

  Currently, the method considers only properties that are generated by the global product data manager for commercial characteristics (OCD) of the article, which is represented by the implicit instance.

  The return value corresponds to that of *getPropChoistList2()* with the following changes/extensions:

  – If the method is called for a property that is not generated by the global product data manager, the return value is of type *Void*.

  – The language-specific description of the value (element 2 of a single value specification) is a language–text mapping (see section 1.4).

  – For values that are not contained in the current choice list of the implicit instance, the flag (bit) with the value 2 is set in the status (element 3) (see *setPropChoiceList()*, section 2.3).

[36] The semantics and purpose of this method differ significantly from method *getPropGroupDescriptions()*, see 2.5.

3 Other aspects

3.1 Options

Properties of types `"Y"` and `"YS"` can be used to implement *options*, i.e. properties that can, but does not have to be evaluated by the user. For this purpose, a special value is added to the choicelist, typically `@VOID` resp. `"VOID"`, with a corresponding text resource (English e.g. "not specified").

However, this is transparent for the clients (e.g. property editors), i.e., this special value appears as a completely "normal" value from the choice list.

3.2 Restrictable OCD properties

Restrictable OCD properties either are realized with OFML properties of types `"Y"` resp. `"YS"` or (in the case of numeric characteristics) with properties of type `"N"` with a choice list of type `@FixedSingleV`.

These properties can have the state "not (yet) specified". In the case of symbolic restrictable properties, this state is represented by the reserved special value `@UNSPECIFIED` resp. `"UNSPECIFIED"` with a corresponding text resource (element 9 in the return Vector of method *getPropSpec()*, section 2.7). In the case of numerical restrictable properties, this state is represented by a value of type *Void*, but element 10 in the return Vector of method *getPropSpec()* contains a language-specific description that corresponds to the text resource for `@UNSPECIFIED`.

Depending on the setting in the product data, the choice list of restrictable properties, in addition to the actual values, also can contain the special value which enables the user to explicitly set the state "not specified". For elements 1 and 2 in the return Vector of methods *getPropChoiceList()* and *getPropChoiceList2()* (see section 2.7) the same statements apply as above for elements 9 and 10 in the return Vector of method *getPropSpec()* (current property value).

Similar to options (see previous section), the subject of restrictable properties is transparent for the clients (property editors), too.

3.3 Empty choice lists

From a conceptual point of view, the choice list of a property must not be empty. However, this may happen due to problems or errors in the product relationship knowledge. In such a case and insofar no value has been explicitly assigned to the property[37], the standard implementation of method *getPropSpec()* (section 2.7) returns a language-specific text in element 10 that describes this state (English e.g. "no value available").

3.4 Application callbacks

- *::ofml::app::isPropTypeSupported(pPType(String))* → *String* **extended**

  The function returns `"1"` (true), if the application supports the property type specified in the parameter, otherwise `"0"` (false).

  An application whose property editor has been changed over to the new interface returns `"0"` for all property types of the old interface.

- *::ofml::app::getPDLanguage(pPID(Symbol))* → *String* **new**

  The function returns the language to be used for product data texts and property-related texts for article instances that belong to the OFML program (series) that is specified in the parameter.

  The language is determined by comparing the language references currently set in the application by the user and the languages supported by the package currently installed for the OFML program.

[37] *getPropValue2()* returns a value of type *Void*

If none of the languages supported by the OFML package corresponds to the language references currently set by the user, the first language from the list of languages supported by the package is used.

The call of this application callback is encapsulated by following OFML methods:

- Default implementation of function

  getPDLanguage() → *String*

  of interface *Article* [article].

- *OiPlanning::getPDLanguage(pArg(Any))* → *String* | *Void*

  The method returns the language to be used for product data texts and property-related texts for article instances that belong to the specified OFML program or for the specified article instance.

  Parameter *pArg* may be the identifier of an OFML program (*Symbol*), a ProgInfo object (*OiProgInfo*) or an article instance. (In case of an invalid parameter return value is of type *Void*.)

  This implementation uses application callback *::ofml::app::getPDLanguage()* passing the specified program identifier resp. the ID of the specified ProgInfo object[38] resp. the OFML program of the specified article instance[39].

[38] according to method *OiProgInfo::getID()*
[39] according to method *getProgram()* of interface *Article*

A Alphabetical index of methods

Preliminary remarks:
The obsolete methods (section B) are not listed here.
New methods are marked with (N).

B Obsolete methods

The following methods (in alphabetical order) are no longer (officially) included in the new specification of the interface. However, for reasons of backward compatibility, they are retained in the implementation.

- *getChLPropValueText(pPKey(Symbol), pPValue(Any), pLanguage(String))* → *String* | *Void*

  The method returns the language-specific description of the specified value of the property with the given key in the specified language.

  The return value is of type *Void* if no description can be provided.

  The method is called during the determination of the language-specific description of the current value of a property with a choice list, see method *getPropSpec()*[40].

- *getProperties()* → *String*

  Replaced by *getPropSpecs()*.

- *getPropertyDef(pKey(Symbol))* → *Any[]*

  Replaced by *getPropDef2()*.

- *getPropertyKeys()* → *Symbol[]*

  Renamed *getPropKeys()*.

- *getPropertyPos((pKey(Symbol))* → *Int* | *Void*

  Renamed *getPropPos()*.

- *getPropClassDescriptions(pLanguage(String))* → *Any*

  Renamed *getPropGroupDescriptions()*.

- *getPropState(pKey(Symbol))* → *Int*

  Replaced by *getPropState2()*.

- *getPropValue(pKey(Symbol))* → *Any*

  Replaced by *getPropValue2()*.

- *getVisiblePropValues()* → *Void*

  Replaced by *getPropChoiceList()*.

- *setPropPosOnly(pKey(Symbol), pPos(Int))* → *Int* | *Void*

  Renamed *setPropPos()*.

- *setPropState(pKey(Symbol), pState(Int))* → *Void*

  Replaced by *setPropState2()*.

- *setupProperty(pKey(Symbol), pDef(Any[5]), pPos(Int))* → *Void*

  Replaced by *setupProperty2()*.

[40] Actually, this method currently no longer has any significance. It was introduced mainly to solve performance problems with articles in an outdated and no longer supported product data format. In the OFML base classes the method has an empty implementation and, thus, no effect.

C Modification history

Version 2.9 (2023-09-14):

- Clarification regarding method *setupPropert2()* (section 2.1):
  If there is already a property registered with the specified key but a different definition, the method call has no effect.
- Method *getPropChoiceList()* (section 2.3) now accepts an additional optional parameter indicating whether the method should be called if the choice list is defined indirectly by specifying the name of a method.
  Default (backward compatible behavior) is 1 (yes).

Version 2.8 (2023-05-04):

- Clarifications regarding the standard implementation of method *setPropValue()* in section 2.4.

Version 2.7 (2023-03-17):

- Clarifications in the specification of method *getPropGroupDescriptions()* in section 2.5.

Version 2.6 (2022-09-14):

- Clarification on the handling of invalid value range definitions when passing to method *setPropRanges()* (section 2.3).
- Added missing specification of method *checkPropValue()* in section 2.4.

Version 2.5 (2022-05-02):

- Refering to function *getPDLanguage()* of interface *Article* in section 3.4.

Version 2.4 (2021-12-22):

- Method *getPropInfo()* is now decribed in section 2.6.
- Method *getChLPropValueText()* is now declared as obsolete (and described in appendix B).
- The procedure for determining the language-specific description of the current value of a property with a choicelist of type `@FixedSingleV` is now described in section 2.7 with method *getPropSpec()* (instead of *getChLPropValueText()*.

  In doing so, the procedure has been corrected: a text specified in the choicelist specification for the value takes precedence over any text provided by *getChLPropValueText()* or by a text resource.

Version 2.3 (2021-10-12):

- Small correction and more detailed description regarding the return structure of methods *getPropSpec()* and *getPropChoiceList2()*.
- Clarification on the use of method *getPropTitle()*.

Version 2.2:

- First english version.