EasternGraphics
visualize your business

The OFML Interfaces *Article* and *CompositeArticle*

Document version 1.3

Thomas Gerth, EasternGraphics GmbH (Editor)

October 21, 2024

Contents

References

[oam] OAM – OFML Article Mappings (OFML Part VI). Specification version 1.0.
Industrieverband Büro und Arbeitswelt e. V. (IBA)

[ocd] OCD – OFML Commercial Data (OFML Part IV). Specification version 4.3.
Industrieverband Büro und Arbeitswelt e. V. (IBA)

[oex] OEX – OFML Business Data Exchange (OFML Part VII). Specification version 3.1.
Industrieverband Büro und Arbeitswelt e. V. (IBA)

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

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

[rec20] Code List Recommendation 20 – Codes for Units of Measure Used in International Trade.
Revision 17 (Annexes I to III), 2021.
The United Nations Economic Commission for Europe (UNECE)
(https://unece.org/trade/uncefact/cl-recommendations)

The documents (except [rec20]) are available at the Download Center of EasternGraphics

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

in the category *OFML Specifications*.

1 Introduction

The interface *Article* defines all functions that must be implemented by a type whose instances represent a commercial article.
(In the following, instances representing an article are also referred to as *article instances*.)

The specifications in this document replace resp. update the interface specification in [ofml]!

A *composite article* consists of a fixed or variable number of sub-articles. A sub-article of a composite article can also be a composite article itself[1].
The interface *CompositeArticle* defines the functions that must be implemented by a type whose instances represent a composite article. It extends the interface *Article*, i.e. the corresponding type must also implement all the functions of interface *Article*!

Functions that are related in content are combined into a method group and specified in corresponding subsections of sections 2 (interface *Article*) and 3 (interface *CompositeArticle*).

Appendix A contains an alphabetical index of the functions.

Base type *OiPlElement* has a default implementation for all functions of interface *Article*. (Where necessary/helpful, the behavior of the default implementation is explicitly described.)
However, base type *OiPlElement* cannot be used directly for an article instance because it does not provide a generic geometry creation. For this, see e.g. base type *OiOdbPlElement* derived directly from *OiPlElement*.

Base type *OiPart* is conceptually not intended for article instances. Since types derived from it have nethertheless been used for this purpose in the past, *OiPart* also has a default implementation for all functions.

The implementation of function *isCat()* (interface *MObject*) in the base types *OiPlElement* and *OiPart* returns value 1 (true) for the interface category `@IF_Article`.
However, in derived classes *isCat()* may be overwritten and then return value 0 for the category `@IF_Article`, i.e. this instance is then not to be considered an article instance. For clients this means that in this case no methods of this interface may be called on this instance[2].

Appendix B contains specifications of predefined standard categories for article instances.

[1] The underlying software design pattern is *Composite*.
[2] If methods are called nevertheless, in the simpler case this only leads to a poorer performance, but in the more severe case this can also lead to an incorrect behavior.

2 Interface *Article*

Functions that have not (yet) been described in the interface specification in [ofml] are marked with **new**.

Functions whose specification has been modified compared to [ofml] are marked with **mod**.

Some functions have a parameter *pLanguage* of type *String* which can be used to specify the desired language for textual components of the return value. The language must be specified as a two-character abbreviation according to ISO 639-1, e.g.: `de` (German) and `en` (English).

2.1 OFML program

- *getProgram()* → *Symbol*

  The function returns the ID of the OFML program to which the implicit instance belongs.

  The program defines, among other things, which instance of *OiProgInfo* registered in the global planning instance (base type *OiPlanning*) is used to perform general program-related tasks. Furthermore, the program defines which product database (instance of *OiProductDB*) registered in the global product data manager (base type *OiPDManager*) contains the commercial data for the article represented by the implicit instance.

  Conceptually, this function belongs to the interface *Base*[3]. However, due to its use when determining the relevant product database (see above), the program ID has a special meaning for article instances.

  The default implementation of interface *Base* delegates the request to the next planning element (instance of *OiPlElement*) higher up in the object hierarchy.

  The default implementation in *OiPlElement* is based on a member variable which is initialized during the creation and initialization of the article instance (see also appendix E).

2.2 Initialization and article codes

- *setArticleSpec(pSpec(String))* → *Void* **mod**

  The function assigns an article number to the implicit instance.

  The article number is an alphanumeric code that uniquely identifies a manufacturer's product (article) within the leading production and planning system (PPS). If it is a configurable article, the article number is also referred to as the *basic article number*, in contrast to a (possibly) additionally specified extended final article number which identifies the fully configured product (see *setXArticleSpec()* below).

  As a result of the function, the instance should have the characteristics (properties) that are defined in the relevant product database[4] for the initial (basic) configuration of the article.

  Whether the corresponding geometric representation is also available in the result of the function depends on the implementation in the specific type used. For reasons of performance, the generation of the geometric representation can be delayed until the subsequent call of function *setXArticleSpec()* (see below), which assigns a (possibly empty) variant code to the implicit instance.

  > For example, the implementation in the base type *OiOdbPlElement* generates only the initial properties using the method *setupProps()* of the global product data manager (base type *OiPDManager*).

  The function has no effect if it is called between the persistence rules `START_EVAL` and `FINISH_EVAL` for the global planning instance.

[3] and will be described there when the OFML specification in [ofml] is revised

[4] The relevant product database is determined based on the program ID of the implicit instance, see 2.1.

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

  The function returns the (base) article number of the article represented by the implicit instance or a value of type *Void* if there is no article specification for the implicit instance.

  If the result of the function is a value of type *Void*, no entry is created for the instance in the basket structure of the application.

  The default implementation delegates the request to the method *object2Article()* of the global product data manager (base type *OiPDManager*), passing the implicit instance as the argument.

  For reasons of performance, concrete types whose instances represent an article should overwrite this implementation and return the value of a corresponding member variable to which the article number is assigned that was passed to function *setArticleSpec()* (see above). This principle is used, for example, in the implementation in base type *OiOdbPlElement*.

- *getArticleParams()* → *Any* **mod**

  The function returns the parameters of the implicit instance, which are to be used in addition to the type of the instance in order to determine the article number.

  The function is relevant only for types that do not have their own implementation of method *getArticleSpec()* using a corresponding member variable (see above), and whose instances can represent different articles. Function *getArticleParams()* is then called from method *object2Article()* of the global product data manager (base type *OiPDManager*).

  Return value is a Vector with the parameter values or a String that already contains the parameter values converted into the respective persistence format. If no parameters are required to determine the article number, the function returns a value of type *Void*.

  The default implementation returns a value of type *Void*.

  The implementation in base type *OiPlElement* delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance as the argument. The global product data manager then uses external mapping tables to determine the parameters. The mapping tables are standardized in OFML Part VI (OAM) [oam].

- *setXArticleSpec(pType(Symbol), pSpec(String))* → *Void* **mod**

  The function assigns an article specification of the specified type to the implicit instance.

  The following specification types are defined:

  `@Base`
  : Basic article number: unique identifier of the manufacturer for the article without reference to a specific design/configuration.

  `@VarCode`
  : Manufacturer-specific variant code: describes the specific design/configuration of the article.

  `@OFMLVarCode`
  : Manufacturer-independent variant code, also called the *OFML variant code*: describes the specific design/configuration of the article.

    The OFML variant code should be used by OFML applications to restore saved article configurations (see section. 2.6). Its structure is described in appendix C.

  `@Final`
  : Manufacturer-specific final article number: identifies the article and describes its specific design/configuration.

    > Normally, the final article number is composed of the basic article number and the variant code. However, this depends on the underlying product data system.

  If an article specification of type `@Base` is passed, the function behaves like function *setArticleSpec()*, see above.

The commercial initialization of an article instance is done by immediately successiv calls to *setArticleSpec()* (or *setXArticleSpec()* with specification type `@Base`) and *setXArticleSpec()* with specification type `@VarCode`.

The passed variant code can be an empty string. The article instance will then keep the initial (basic) configuration created during *setArticleSpec()*.

The passed variant code can be partially determined, i.e. it can only code the properties that are to be evaluated differently from the basic configuration.

If a variant code or a final article number is passed that does not match the article represented by the implicit instance, the instance retains the configuration generated up to that point or only part of the properies coded in the passed specification are re-evaluated.

When assigning a variant code or a final article number, the implicit instance in the result of the function also has the geometric representation corresponding to the created article configuration.

- *getXArticleSpec(pType(Symbol))* → *String* | *Void* **mod**

The function returns the specification of the required type for the article represented by the implicit instance.

If there is no article specification of the required type for the implicit instance, the function returns a value of type *Void*.

The possible specification types are described with function *setXArticleSpec()* (see above).

If an article specification of type `@Base` is passed, the return value corresponds to that of function *getArticleSpec()* (see above).

The standard implementation delegates the request to method *getArticleSpec()* for specification type `@Base` and to equal named method of the global product data manager (base type *OiPDManager*) for all other specification types, passing the implicit instance and the specification type as arguments.

2.3 Other State

- *getObjState(pStateType(Symbol))* → *Any* **new**

The function returns the current value of the implicit instance regarding the state of the specified type.

This function is already defined for the basic interface *Base*. The default implementation of interface *Article* overrides the implementation inherited from interface *Base* as follows:
It handles the state types `@OI_UpdateState` and `@OI_ConsistencyState` (see below) and calls the inherited implementation for all other state types[5].

The state type `@OI_UpdateState` refers to the update state that defines the usability of the currently installed product database.
The possible values (of type *Symbol*) for the update state and the functions used during the update of article instances are specified in section 2.6.

The state type `@OI_ConsistencyState` refers to the consistency state containing the result of the last call of method *checkConsistency()*, see section 2.7.
Immediately after creating the article instance, the consistency state is undefined, which is expressed with a value of type *Void*.
The consistency state is stored in a member variable of the article instance and is used as the result of *checkConsistency()* (instead of actually performing the check) if the article instance has not yet been updated resp. could not be updated (see state type `@OI_UpdateState` above) after loading from a dump representation (see section 2.6).

[5] The default implementation of interface *Base* handles state types related to flags that are managed in the OFML runtime environment for each object.

- *isUp2Date() → Int* **new**

  The function returns the value 1 (true) if the currently installed product database may be to be used to change the configuration of the implicit article instance.

  The result is the current update status according to function *getObjState()* (see above) for state type `@OI_UpdateState`: The return value is 1 if the current update status is `@Up2Date`, otherwise 0.

- *setObjState(pStateType(Symbol), pValue(Any)) → Void* **new**

  The function assigns a new value to the implicit instance regarding the state of the specified type.

  This function is already defined for the basic interface *Base*. The default implementation of interface *Article* overrides the implementation inherited from interface *Base* as follows:
  It handles the state types `@OI_UpdateState` and `@OI_ConsistencyState` (see function *getObjState()* above) and calls the inherited implementation for all other state types.

  In the case of a changed value for state type `@OI_UpdateState`, the default implementation performs the following actions in addition to assigning the value to a member variable:

  – If the new state is `@Migratable` or `@Invalid` (see appendix D), all currently editable properties are deactivated (see method *invalidateProperties()* in the interface *Property*). Thus, the article instance can not (anymore) be configured in these states.

  – If the new state is `@Up2Date` (see appendix D), an event of type `@ArticleUpdated` is sent to the global ChangeManager (base type *OiChangeManager*), with the implicit instance as the publisher and the old state as the event argument.

- *getAddStateCode(pDomain(Void | Symbol)) → String | Vector* **new**

  The function returns a code that describes the state of the internal variables of the implicit instance, which differ from the state immediately after the creation and commercial initialization of the article instance[6].

  This code is primarily required and retrieved by OFML applications that store the state of article instances in a persistent basket structure, see section 2.6.

  The code returned by this function should not contain any representation of the current article configuration, as this is already returned by function *getXArticleSpec()* (see section 2.2).

  If the parameter *pDomain* is a value of type *Void*, the complete additional state code has to be supplied. The code is then split into separate sections for different domains and the return value is a vector of vector pairs, each describing the state of a specific domain:

  1. domain (Symbol)
  2. code (String)

     sample: `[[@Domain1, "Code1"],[@Domain2, "Code2"], ...]`

  OFML applications use this variant of the function call when creating a persistent basket representation of the article instance.

  If a client has knowledge about specific domains and wants to get the additional state code only for a specific domain, this can be specified in parameter *pDomain*. In this case, the return value is a string containing only the code for the requested domain[7], or an empty string if the domain is not supported by the type of the implicit instance.

  The default implementation returns an empty vector if the parameter *pDomain* is a value of type *Void*, otherwise an empty string.

[6] Therefore, this code is called the additional state code.
[7] e.g. `"Code1"` for domain `@Domain1`

When this function is overridden in a derived class (together with associated function *setAddStateCode()*, see below), the following rules should be observed:

- If the parameter *pDomain* is a value of type *Void*, the inherited implementation must be called first. Then the own domain section has to be added (if any).
- If the parameter *pDomain is not* a value of type *Void*, the corresponding code is returned if the domain is supported by the class. Otherwise, the result of the inherited implementation has to be returned.
- To avoid naming conflicts with domain identifiers, it is recommended to use the name of the class itself for the section that the class inserts into the code.
  The domain `@ChildProps` is reserved for Meta types.

If necessary, a class/implementation can define multiple domains.

- *setAddStateCode(pDomain(Void | Symbol), pCode(Vector | String))* → *Void* **new**

The function assigns the specified additional state code to the implicit instance.
This code describes the state of the internal variables of the implicit instance, which differ from the state immediately after the creation and commercial initialization of the article instance.

If the parameter *pDomain* is a value of type *Void*, the complete additional state code is passed in the form of a vector containing information about all relevant domains. For the structure of this vector see specification of function *getAddStateCode()* above.

If the parameter *pDomain is not* a value of type *Void*, only the code (String) relevant to this domain is passed.

The client is expected to use the same value for parameter *pDomain* as it did when calling *getAddStateCode()* to get the passed code.

OFML applications use the function with a value of type *Void* for parameter *pDomain* when restoring an article instance from a saved basket representation. In doing so, the call is made after the saved commercial configuration was restored by calling *setupConfiguration()* (see section 2.6)[8].

The default implementation is empty, i.e. no actions are performed.

When this function is overridden in a derived class (together with associated function *getAddStateCode()*, see above), the following rules should be observed:

- If the parameter *pDomain* is a value of type *Void*, the section to be handled by the class must first be extracted from the passed code. Then the inherited implementation must be called passing the (possibly) truncated vector. Finally, the code for the own (extracted) domain must be processed (if any).
- If the parameter *pDomain is not* a value of type *Void*, the passed code will be processed if the domain is supported by the class. Otherwise, the inherited implementation must be called.

[8] Therefore, analogous to the definition for function *getAddStateCode()* above, the passed code should not contain any representation of configurable properties of the article.

2.4 Product data

Preliminary remark:
Conceptually, the product data also includes feature descriptions and variant texts. Since numerous functions are available for retrieving this information, these are dealt with in a separate section (2.5).

Different article instances can represent the same configuration of an article. Conceptually, the functions described in this and the following section should deliver the same results for the same **article configuration** (taking into account any language parameters that may be present).

Two article instances represent the same configuration if the following information match:

- the OFML program ID, see function *getProgram()* (section 2.1)
- the (basic) article number, see function *getArticleSpec()* (section 2.2)
- the OFML variant code, see function *getXArticleSpec()* (section 2.2)

Applications can use this information to generate a key for *product data caches* in order to minimize the number of method calls and thus improve performance. If such a cache is also used for price information, the key must additionally encode the price date, see function *getPriceDate()* (section 2.7).

- *getArticlePrice(pLanguage(String), ...)* → *Any[]* | *Void* **mod**

  The function returns price information for the implicit article instance.

  Parameter *pLanguage* specifies the language to be used for textual elements of the price information.

  If an additional, optional parameter of type *String* is specified, it specifies the desired currency. However, the function does not have to provide the price information in this currency (e.g. if the underlying product database cannot provide prices in this currency). In this case, the client may have to convert to the desired currency itself using exchange rates.

  If no price information is available for the article in the product data, the function returns a value of the type *Void*.
  Otherwise the return value is a List containing the individual *price components* resp. the final price.

  Each entry in the return list is a Vector of 3, optionally 5 elements:

  1. a description (*String*) specifying the type or the reason for the existence of the price component, e.g. the reason for a surcharge
  2. the sales price of the price component (*Float*)
  3. the purchase price of the price component (*float*)
  4. (optional) the identifier of the variant condition of the price component (*Void* | *String*)
  5. (optional) the factor applied to the amount from the price database when calculating the amount of the price component (*Void* | *Float*)

  The first entry represents an exception since it contains the applied currency (*String*) instead of the prices: the second element specifies the currency of the sales price and the third element specifies the currency of the purchase price. (The other elements in this entry have no meaning.)

  The last entry in the list specifies the (accumulated) final price. The optional entries in between specify the individual price components (base price, surcharges, discounts, etc.). If such a price component contains the description `"@baseprice"` in the first element, it is explicitly marked as a base price.

  Elements 4 and 5 are only included in the return structure if these aspects are relevant during price determination in the relevant product database[9].

  Elements 4 and 5 in the entry for the final price are relevant only if no other price components are included in the return list and if the final price is equal to the base price.

[9] The relevant product database is determined based on the program ID of the implicit instance, see 2.1.

If the final price amount for a price type (sales price vs. purchase price) is 0.0 and if an empty string for the corresponding currency is specified in the first list entry for this price type, this means that there is no information on this price type for the article in the product data.

If the relevant product database supports validity periods for price components when determining the price, the date is used that is returned by method *getPriceDate()* for the implicit instance (see section 2.7).

If no price information for the article is available in the product data on this date, the function returns the following value:

```
@(["@invalid_date", NULL, NULL], ["invalid price date", NULL, NULL])
```

The resource `@invalid_date` in the first list entry may be qualified with a concrete package identifier. The description of the state in the second list entry should be specified in the language requested via parameter *pLanguage*.

In this case, the OFML runtime environment (additionally) has to ensure that function *checkConsistency()* (see section 2.7) returns a corresponding inconsistency for the implicit instance.

The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing as arguments the implicit instance, the requested language and the desired currency (if any).

- *getArticleText(pLanguage(String), pForm(Symbol))* → *String[]* | *Void* **mod**

  The function returns a textual description of the requested form in the specified language for the article represented by the implicit instance.

  Possible values for parameter *pForm* are:

  `@l` Long description (article long text)

  The article long text should describe all essential fixed (non-configurable) features of the article[10].
  The article long text usually consists of multiple lines.

  `@s` Short description (article short text)

  The (rather technically oriented) article short text is often an abbreviated version of the long text.
  The article short text should contain only a single line.

  `@pn` Product name

  The (rather marketing-driven) product name can be used in user-oriented product views (instead of the article short text).
  The product name usually consists of a single line.

  The return value is a List of strings containing the individual lines of the description or a value of type *Void* if no description of the requested form is available in the specified language.

  The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance, the required language and the requested form as arguments[11].

[10] A description of the current values of the changeable/configurable properties of the article is provided by function *getArticleFeatures()* (see section 2.5).

[11] The default implementation in *OiPDManager* in turn delegates to the equal named method of the relevant product database (base type *OiProductDB*).

- *getArticleClassifications(pLanguages(String[] | Void))* → *Any* | *Void* **new**

  The function returns information about the classification of the article represented by the implicit instance.

  Paramter *pLanguages* can be used to specify the languages (codes according to ISO 639-1 Alpha 2) for which language-specific descriptions of the classes are to be supplied. If the parameter has a value of type *Void*, no language-specific descriptions are supplied. If the parameter contains an empty sequence (*List* or *Vector*), all language-specific descriptions stored in the product database are supplied.

  If no classification information is available for the article, a value of type *Void* is returned, otherwise a (non-empty) *Vector* of classification information items. One *classification information item* describes a classification according to a specific classification system.

  A classification information item is a *Vector* consisting of the following elements:

  1. Name/identifier of the classification system, without version information or the like (*String*).

     The following identifiers are currently predefined for cross-industry and cross-company standards:

     `ECLASS` Classification according to standard ECLASS[12]

     `UNSPSC` Classification according to standard UNSPSC[13]

     Identifiers for other classification systems start with the character `@`. Identifiers for manufacturer-specific classification systems consist of the character `@` followed by the commercial manufacturer ID.

  2. Qualifier of the system, e.g. the version (*String*).

     The string can be empty if no special qualifier is stored in the product data.

     Semantics and syntax of the qualifier depend on the data format of the relevant product database. For example, for the format version 4.3 of OCD [ocd] the following applies:

     - With standard ECLASS, the qualifier is the version number in the format `x.y`.
     - With a manufacturer-specific system, the qualifier is the part from the OCD system identifier after the manufacturer ID and the underscore.

  3. Identifier of the class (*String*).

  4. Language-specific descriptions of the class (*Any[]*| *Void*).

     If parameter *pLanguages* has a value of type *Void*, then this element also has a value of type *Void*. Otherwise, the element is a *language–text mapping* consisting of a sequence (*Vector* or *List*) of zero or more elements, where each element again is a *Vector* of two elements:

     1. language code according to ISO 639-1 Alpha 2 (*String*)
     2. Text/description in the corresponding language (*String*)

     The language code must consist of two lowercase letters. Combinations of letters that do not correspond to any officially registered code are not explicitly permitted, but should not lead to errors when processed by the client.
     A language–text mapping must not contain two entries with the same language code!

     If parameter *pLanguages* is an empty sequence, the language–text mapping contains all language-specific descriptions stored in the product database.
     Otherwise, the mapping contains the language-specific descriptions for the languages requested in the parameter. (If no description is stored for the class for a requested language, the ID of the class itself is used.)

[12] www.eclass.eu
[13] www.unspsc.org

The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance and the required languages as arguments[14].

- *getPDInfo(pLanguage(String)) → Any[] | Void* **new**

  The function returns additional product information about the article represented by the implicit instance.

  Parameter *pLanguage* specifies the language to be used for textual information.

  Return value is a Vector of information items or a value of type *Void* if there is no additional product data information.

  The content and the order of the information items depends on the relevant product database[15].

  The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance and the required language as arguments[16].

  Product databases based on the commercial data format standardized in OFML Part IV (OCD) [ocd] must provide at least the following two information items:

  1. commercial manufacturer ID (*String*)
  2. commercial series ID (*String*)

- *getOrderUnit(pLanguage(String | Void)) → String* **new**

  The function returns the order unit for the article represented by the implicit instance.

  If parameter *pLanguage* is a value of type *Void*, the language-independent code according to UNECE Recommendation 20 is returned [rec20], e.g. `C62` – *piece*, `MTR` – *meter* and `MTK` – *square meter*.

  If parameter *pLanguage* is a language code (type *String*), the unit name will be supplied in the required language.

  If the product data does not contain information about the order unit of the article, the standard unit `C62` resp. "piece" will be assumed and returned.

  The default implementation delegates (first) to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance as an argument.
  If parameter *pLanguage is not* a value of type *Void*, then the name in the required language is determined for the most common units using a text resource for the language-independent code[17].
  (If no text resource is available, the language-independent code itself is returned.)

- *getArticleAttribute(pAttr(Symbol)) → Any* **new**

  The function returns the value for the requested attribute for the article represented by the implicit instance.

  Currently the following attributes are defined:

  `@Discountable`
  : The attribute of type *Int* specifies whether discounts can be applied to the possibly specified purchase price of the article or not. Value 0 (no) means that *no* discounts from the purchase price are permitted for this article, deviating from the general conditioning of the manufacturer.

[14] The default implementation in *OiPDManager* in turn delegates to the equal named method of the relevant product database (base type *OiProductDB*).
[15] The relevant product database is determined based on the program ID of the implicit instance, see 2.1.
[16] The default implementation in *OiPDManager* in turn delegates to the equal named method of the relevant product database (base type *OiProductDB*).
[17] see global function *oiGetOrderUnitDescr()*

If an undefined attribute is requested or if the product data for the article does not contain information about the attribute, a value of type *Void* is returned.

The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing as the first argument the object supplied by method *getArticleObj()* (see above) of the implicit instance, and as the second argument the requested attribute.

Notes on the functions *getPDInfo()*, *getOrderUnit()* and *getArticleAttribute()*:

- These functions were introduced in this order with time intervals and all serve to retrieve specific information about an article. From a conceptual point of view, a uniform, general function would be desirable, so that a new function does not have to be introduced with every new relevant information. This was addressed with the introduction of function *getArticleAttribute()*. However, for reasons of downward compatibility, the other two functions were retained.
- For performance reasons, it is often desirable for clients to be able to retrieve several pieces of information of different types with one function call. However, since the needs of different clients differ with regard to the scope of the information to be retrieved, such a function cannot be defined generally in the context of the interface *Article*. Therefore, this must be done in specific OFML libraries if required.

2.5 Features and variant text

- *getArticleFeatures(pLanguage(String | Void) → Any* **mod**

  The function returns a description of the currently defined configurable properties (features) of the article represented by the implicit instance.

  The function either returns a value of type *Void* if there is no feature description for the implicit instance, or a list of Vectors:

  1. property (*String*)
  2. current property value (*String* | *Void*)

  The content of the vector elements depends on the parameter *pLanguage*:

  – If the parameter has a value of type *Void*, language-independent identifiers for properties and values are returned. (Numeric values are converted into type *String* for this purpose[18]).

    This form of function call can be used by OFML applications to generate a description of the article configuration that can be exported to an external production and planning system (PPS) for order processing.

  – If the parameter contains a language code (type *String*), the names of properties and values are supplied in the required language. The value element can be a value of type *Void*, see below.

    This form of function call is used by OFML applications to generate the so-called **variant text**. The following rules apply to this:

    * Each vector in the return list creates one line of the variant text.
      (Therefore, the strings in the elements for property and value themselves must not contain any characters for a line break[19].)
    * If the value element is of type *Void*, the property element contains the complete line for the variant text.
      This can be used for multi-line descriptions of features (which are then realized by immediately consecutive entries in the return list).

[18] see constructors of type *String*
[19] If they do, they are replaced by the applications with a space character.

* If both vector elements contain a string, the line of the variant text results from the concatenation of both strings, separated by the string ": ".

In both forms, the return list contains only currently visible/valid configurable properties with a defined value. An exception are not selected valid optional properties with a special description for the unselected state: these are included in the return list if parameter *pLanguage* contains a language code.

On the other hand, certain defined and visible configurable properties can be omitted in the return list for the variant text. This can be the case if the feature is indirectly described by dependent features, see e.g. the text control code in the OCD property table [ocd].

The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance and the desired language as arguments[20].

- *getArticleFeatures2(pLanguage(String)* → *Any* **new**

  The function returns an alternative description of the currently defined configurable properties (features) of the article represented by the implicit instance.

  The function either returns a value of type *Void* if there is no feature description for the implicit instance, or a list of Vectors, each describing one property:

  1. language-independent identifier of the property (*String*)
  2. language-specific name of the property (*String*)
  3. current property value (*String* | *Void*)[21]
  4. language-specific description of the value (*String*)

  The language-specific elements are returned in the language specified by parameter *pLanguage*.

  The return list contains only currently visible/valid configurable properties with a defined value. An exception are not selected valid optional properties with a special description for the unselected state: these are included in the return list with a value of type *Void* in the 3rd element.

  In contrast to function *getArticleFeatures()* (see above), for property values with a multi-line description (variant text) only the first line is used/returned in the 4th element (see also the text control code in the OCD property table [ocd]).

  The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance and the desired language as arguments.

- *getAllArticleFeatures(pLanguage(String)* → *Any* **new**

  The function returns a description of all current property assignments (features) of the article represented by the implicite instance.

  In contrast to functions *getArticleFeatures()* and *getArticleFeatures2()* (see above), this can include internal properties that are not visible/configurable for the user.

  The function either returns a value of type *Void* if there is no feature description for the implicit instance, or a list of Vectors, each describing one property:

  1. language-independent identifier of the property (*String*)
  2. language-specific name of the property (*String*)
  3. current property value (*String* | *Void*)[22]
  4. language-specific description of the value (*String*)
  5. Flag (*Int*) indicating whether the property is visible/configurable (1) or not (0).

[20] The default implementation in *OiPDManager* in turn delegates to method *getPropDescription()* of the relevant product database (base type *OiProductDB*).
[21] Numeric values are converted into type *String*.
[22] Numeric values are converted into type *String*.

The language-specific elements are returned in the language specified by parameter *pLanguage*.

The transmission of internal properties is optional, i.e. it depends on the implementation and the data format of the relevant product database[23].

If there is no language-specific name or no language-specific description for the value of an internal (non-visible) property in the product data, the 2nd element contains an empty character string resp. the 4th element is identical to the 3rd element.

The return list contains only properties that currently have a defined value. An exception are not selected valid optional properties with a special description for the unselected state: these are included in the return list with a value of type *Void* in the 3rd element.

The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance and the desired language as arguments.

- *getArticleFeaturesDescr(pType(Symbol), pLanguage(String)* → *Any* **new**

  The function returns a description of the properties (features) of the article represented by the implicit instance.

  The desired form of the description is specified in the parameter *pType*.

  The language-specific elements are returned in the language specified by parameter *pLanguage*.

  Currently, the following description types are defined:

  `@Text`
  : The description corresponds to the result of the function call `getArticleFeatures(pLanguage)`

  `@AllIDs`
  : The description corresponds to the result of the function call `getAllArticleFeatures(pLanguage)`

  `@ID_Text`
  : The function either returns a value of type *Void* if there is no feature description for the implicit instance, or a list of Vectors, each describing a currently defined configurable property:

    1. language-independent identifier of the property (*String*)
    2. current property value (*String* | *Void*)[24]
    3. List of *String* pairs (type *Vector*), each specifying one line of the language-specific description of the feature
       The first string is intended for the language-specific name of the property and the second for the language-specific description of the value. For the creation of the corresponding line of the feature description on part of the client the regulations apply which are mentioned with function *getArticleFeatures()* (see above) for the creation of the variant text.

    The return list contains only currently visible/valid configurable properties with a defined value. An exception are not selected valid optional properties with a special description for the unselected state: these are included in the return list with a value of type *Void* in the 2nd element.

    On the other hand, certain defined and visible configurable properties can be omitted in the return list. See also the correspondig comment for function *getArticleFeatures()* (see above).

  The default implementation calls function *getArticleFeatures()* for description type `@Text` and function *getAllArticleFeatures()* for description type `@AllIDs`. For all other description types, delegation is made to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance, the required description type and the desired language as arguments.

[23] The relevant product database is determined based on the program ID of the implicit instance, see 2.1.
[24] Numeric values are converted into type *String*.

2.6 Persistence and update

In principle, the following two forms of a persistent representation of an article can and must be considered:

Dump representation
: In this form, the complete internal state (member variables) of the OFML instance itself, which represents an article, is serialized.

  Applications can use this form to save a complete OFML scene or to write the state of the OFML instance to the clipboard resp. to restore the instance from the state saved in the clipboard.

  When saving and loading a dump, the persistence rules specified in [ofml] are applied.

  The dump format is not standardized[25].

Basket representation
: In this form, all relevant information about the article is saved, which is necessary to be able to (further) process it in order transactions. This also includes information needed to recreate the OFML instance, e.g. if the article is to be reconfigured.

  The following information is required to recreate the OFML instance from a basket representation:

  - the OFML program ID, see function *getProgram()* (section 2.1)
  - the (basic) article number, see function *getArticleSpec()* (section 2.2)
  - the OFML variant code, see function *getXArticleSpec()* (section 2.2)
  - the so-called *AddStateCode*, see function *getAddStateCode()* (section 2.3)

  The basket format is not standardized[26].

The functions specified below in this section are used to implement a general concept for the handling of saved articles in relation to the currently installed product data. These may differ from the product data with which the article was last created resp. reconfigured and saved.

The basic regulations of this concept are:

- Immediately after loading a saved plannng or a saved basket, the articles contained in it have all the characteristics (configuration, texts, price) as at the time of saving.

  Initially, the state of the articles in relation to the currently installed product data is *unknown/undefined* (`Undefined`), because usually the version of the product data used when saving is not known and it must be assumed that the product data has changed in the meantime.

  If the used basket format provides for saving the version of the used product data, after comparing the saved version with the version of the currently installed product data, where appropriate, the application also can set the state of the concerned articles to *up-to-date/updated* (`Up2Date`)[27]. This option does not exist for saved OFML scenes (dump format), because the interface *Article* (currently) does not provide for assigning the product data version to an article instance.

- In order to be able to edit (reconfigure) the article or to use texts and prices from the currently installed product data, the article has to *be updated*.

  After the update, the article is in the state *up-to-date/updated* (`Up2Date`).

  As long as an article is not in this state, it cannot/may not be changed (offer commitment)!

- Before an article is updated, it must be determined whether the article can be updated at all. To this end, it must meet the following conditions:

  1. The article is contained in the currently installed product data in the same commercial series.
  2. The saved configuration of the article can be restored with the currently installed product data, i.e., all properties that can be changed (configured) by the user according to the saved

[25] Applications of EasternGraphics use the FML format.
[26] Applications of EasternGraphics use the OBX format.
[27] The OBX format used in applications of EasternGraphics currently does not provide this option.

configuration can also be configured with the current product data and the saved values of these properties are also valid with the current product data[28].

The manufacturer-independent OFML variant code is recommended as the basis for checking the updatability, see section 2.2 and appendix C.

After the updatability check, the article can be in 3 states:

- `Invalid`, if condition 1 is not met
- `Updatable`, if both conditions are met
- `Migratable`, if condition 1 is met but not condition 2

The updatability check is conceptually a separate step, independent of the actual update[29]. From the user's point of view, however, the applications usually combine both steps into one action.

- *Migration* is a special form of updating that accepts that the saved configuration cannot be restored exactly the same. During the migration, an attempt is made to adopt as many of the saved property values as possible.

  It is up to the OFML applications whether they support the migration, and if so, whether and in what form the user is involved.

Appendix D contains a graphical representation of the possible states and their relationships.

The basic principles described above are supplemented by the following specific provisions:

- *Composite articles* may only be updated in their entirety, i.e., as soon as one of the articles in the compound cannot be updated, all other articles in the composition may not be updated either.
- An article instance that was inserted via *copy/cut and paste* adopts the update state of the original instance.

The following functions are used to implement the concept described above:

- *setupConfiguration(pBaseArticle(String), pArticleCode(String), pCodeType(Symbol), pMigration(Int))* → *Int* **new**

  The function assigns the passed basic article number to the implicit instance and then tries to create/restore the configuration described by the passed article code on the basis of the currently installed product data.

  The product database to be used is determined based on the program ID of the implicit instance, see 2.1.

  Parameter *pCodeType* is used to specify the type of the passed article code. Currently the types `@VarCode` and `@OFMLVarCode` are permitted, see function *setXArticleSpec()* in section 2.2. According to the recommendation in the concept described above, it is preferable to use the manufacturer-independent OFML variant code[30].

  This function is primarily intended for restoring an article instance from a saved basket representation.

  If no product database is installed for the OFML program of the implicit instance or if it does not contain an article with the specified basic article number, the configuration of the implicit instance remains unchanged[31], its update state is set to `@Invalid` and the return value is 0.

  If the configuration can be completely restored, the update state of the implicit instance is set to `@Up2Date` and the return value is 1.

[28] Therefore, this condition is **not** met if a configurable property is not (no longer) contained in the current product data or is not (no longer) configurable, or if a stored value of a configurable property is not (no longer) valid in the current product data. On the other hand, a configurable property added in the current product data does not violate the condition.

[29] This is reflected in the functions specified in this section.

[30] which applies to the applications of EasternGraphics.

[31] Then, the passed basic article number is not adopted either.

If the configuration *cannot* be completely restored and if parameter *pMigration* has the value 0 (false), the configuration of the implicit instance remains unchanged[32], its update state is set to `@Migratable` and the return value is 0.

If the configuration *cannot* be completely restored and if parameter *pMigration* has the value 1 (true), the article instance adopts the partially restored configuration, its update state is set to `@Up2Date` and the return value is 0.

In the case of an update or migration, the basic article number and the variant code are assigned by means of immediately successive calls of methods *setArticleSpec()* and *setXArticleSpec()* on the implicit instance (see section 2.2).

The default implementation delegates to the equal named method of the global product data manager (base type *OiPDManager*).

- *updateConfiguration()* → *Void* **new**

  The function updates resp. migrates the configuration of the implicit article instance based on the currently installed product data.

  The product database to be used is determined based on the program ID of the implicit instance, see 2.1.

  This function is primarily intended for updating or migrating an article instance loaded from a dump representation.

  If the implicit instance is part of a composite article, the default implementation delegates the request to the composite article instance at the highest level in the object hierarchy above the implicit instance.

  Otherwise, the default implementation behaves as follows:

  1. The function has no effect if the update state is `@Up2Date` or `@Invalid`.
  2. If the current update state is `@Undefined`, first the method *checkObjUpdatability()* of the global product data manager (base type *OiPDManager*) is called, passing the implicit instance and `@OFMLVarCode` as the code type. The update state returned by this method is then assigned to the implicit instance, see function *setObjState()* (section 2.3).
  3. If the current update state of the implicit instance is now `@Invalid` or if the update state is `@Migratable` and the user does not want to perform a migration, the function will terminate.
  4. Now the commercial update (`@Updatable`) resp. migration (`@Migratable`) is done by delegation to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance as the argument.
  5. Now method *updateGeometry()* (see below) is called on the implicit instance in order to make adjustments to the geometry, if necessary.
  6. Finally, the update state of the implicit instance is set to `@Up2Date`, see function *setObjState()* (section 2.3).

- *checkUpdatability(pCodeType(Symbol))* → *Symbol* **new**

  The function checks whether the article with the base article number of the implicit instance is contained in the currently installed product data and whether the configuration coded in the article code of the implicit instance can be completely restored with this product data.

  The product database to be used is determined based on the program ID of the implicit instance, see 2.1.

  Parameter *pCodeType* specifies the type of article code to be used. Currently, types `@VarCode` and `@OFMLVarCode` are permitted, see function *setXArticleSpec()* (section 2.2).

  The return value is `@Invalid` if no product database is installed for the OFML program of the implicit instance or if it does not contain an article with the basic article number of the implicit instance.

[32] Then, the passed basic article number is not adopted either.

The return value is `@Migratable` if the product database contains the article with the basic article number of the implicit instance, but the configuration described by the article code of the specified type cannot be completely restored.

The return value is `@Updatable` if the product database contains the article with the base article number of the implicit instance and the configuration described by the article code of the specified type can be completely restored.

The method call does not change the update state of the implicit instance itself.

The default implementation delegates to method *checkObjUpdatability()* of the global product data manager (base type *OiPDManager*), passing the implicit instance and the specified code type.

- *updateGeometry()* → *Void* **new**

  The function updates the geometry of the implicit instance according to its current article configuration.

  The function is called by the default implementation of function *updateConfiguration()* (see above).

  The default implementation is empty.

  The implementation in the base type *OiOdbPlElement* updates the ODB object hierarchy according to the current state of the Hash table of ODB parameters (which may have changed after the previous commercial update resp. migration).

2.7 Consistency check and price date

- *checkConsistency()* → *Int* | *Void* **mod**

  The function checks the consistency and completeness of the implicit instance.

  If necessary, corrections or additions are made or error messages are generated.

  If the consistency and completeness of the instance cannot be determined unambiguously, a value of type *Void* is returned. Otherwise, the return value is 1 (ok) if the instance is consistent and complete, and 0 if not.

  If an *error log* has been created by the higher-level instance that initiated the consistency check of the implicit instance, the error messages should be written to this log, otherwise they can be issued directly to the user by means of global function *oiOutput()*.

  The error log to be used must be retrieved from the global planning instance (type *OiPlanning*) using method *getErrorLog()*.

  The data structure of the error log specified for *checkConsistency()* is a Hash table in which the relevant messages are entered for each article instance. The key for the Hash table is either the order ID of the article instance (see function *getOrderID()* in section 2.8 below) or its absolute hierarchical object name (see function *getName()* of interface *MObject*).

  The value for this key is a list of vectors containg these three elements:

  1. the error message (String)
  2. the absolute hierarchical object name of the instance that reported the error (String)
  3. the name of the method in which the error was detected (String)

  The behavior of the default implementation is as follows:

  - If a hash table for the error log is set up in the global planning instance (type *OiPlanning*) according to the method *getErrorLog()*, inconsistency messages are stored in this table, otherwise the messages are issued to the user by means of global function *oiOutput()*.
  - If the update state of the implicit instance is *not* `@Up2Date` (see section 2.6 and appendix D), the result is the return value of method *getObjState()* of the implicit instance for the state type `@OI_ConsistencyState` (see section 2.3) and the function terminates.

– The commercial consistency check is done by delegation to the equal named method of the global product data manager (base type *OiPDManager*), passing the implicit instance as the argument.

The scope of the commercial consistency check depends on the implementation and the data format of the relevant product database[33]

If the relevant product database supports validity periods for price components when determining prices, it is required to check whether price information is available in the (currently installed) product data for the price date of the implicit instance (see method *getPriceDate()* below).

– After the commercial consistency check, a delegation is made to method *checkObjConsistency()* of the instance of *OiProgInfo*, which is registered in the global planning instance (base type *OiPlanning*) for the OFML program of the implicit instance. In doing so, the implicit instance is passed as an argument.

This can be used to perform additional program-specific checks on the implicit instance.

– The result of the consistency check is stored in the implicit instance by calling method *setObjState()* for state type `@OI_ConsistencyState` (see section 2.3).

- *setPriceDate(pDate(String))* → *Void* **new**

The function assigns the date to be used during price determination to the implicit instance.

The date must be given in the format `YYYYMMDD`. If the date is not passed in this format, the function has no effect.

If the implicit instance belongs to the interface category `@IF_Article`, the default implementation delegates to the equal named method of the instance, which is returned by function *getArticleObj()* (see section 2.4). Otherwise, the default implementation assigns the passed date (if valid) to a member variable of the implicit instance.

- *getPriceDate()* → *String* | *Void* **new**

The function returns the date to be used during price determination for the implicit instance.

If no (valid) date has been assigned to the implicit instance using method *setPriceDate()* (see above), a value of type *Void* will be returned. Otherwise, the return value is a string in the format `YYYYMMDD`.

If the implicit instance belongs to the interface category `@IF_Article`, the default implementation delegates to the equal named method of the instance, which is returned by function *getArticleObj()* (see section 2.4). Otherwise, the default implementation returns the date that was assigned by a previous call of method *setPriceDate()* resp. a value of type *Void* if no (valid) date was assigned before.

2.8 Others

- *getArticleObj()* → *MObject* **new**

The function returns the object whose methods of interface *Article* must be used to get information (text, price) about the article represented by the implicit instance.

The default implementation returns the implicit instance itself.

The function can resp. must be overridden in Meta types in order to delegate to a special (encapsulated) child object. The type of this object must implement the interface *Article*!

Clients must call this function and call the functions described further in this section on the returned object to get the article information for the implicit instance!

[33] The relevant product database is determined based on the program ID of the implicit instance, see 2.1.

- *getPDLanguage()* → *String* **new**

  The method returns the language to be used for product data texts and property-related texts for the implicite instance.

  The default implementation[34] is based on a member variable that is initialized resp. updated at the following times by calling method *getPDLanguage()* on the global planning instance (base type *OiPlanning*)[35] (passing the implicit instance as an argument):

  – on creation of the instance
  – on each change of the OFML program of the implicit instance (see section 2.1) by calling method *setProgram()* (only for instances of type *OiPlElement*)
  – on each call of method *updateProperties()* of interface *Property* [property] for the implicite instance if the update state of the instance is `@Up2Date` (see section 2.6)

- *setOrderID(pID(Symbol))* → *Void*

  The function assigns a unique order ID to the implicit instance.

  The order ID can be used to synchronize a basket structure managed by the application with the article instances in the planning hierarchy (OFML scene). The order ID is then used to assign an article position in the basket to the instance that represents the article in the planning.

  The order ID is assigned to the article instance by the global planning instance (base type *OiPlanning*) immediately after its creation and is not changed for the duration of the instance's existence.

  If the position of the article instance in the planning hierarchy changes due to a cut/paste operation, the order ID from the destroyed instance is transferred to the newly created clone instance.

  The default implementation stores the passed ID in a corresponding member variable.

- *getOrderID()* → *Symbol* | *Void*

  The function returns the unique order ID of the implicit instance.

  If no order ID has been assigned to the implicit instance (see function *setOrderID()* above), a value of type *Void* is returned. (This applies to applications that do not manage a OFML scene.)

  The default implementation returns the current value of the member variable to which the ID is assigned during *setOrderID()*.

- *setCatalogInfo(pInfo(Any[]))* → *Void* **new**

  The function assigns to the implicit instance information about the entry in the catalog of the application that was used when inserting/creating the article instance.

  The information is passed as a List of vectors with two elements:

  1. type of information (*Symbol*)
  2. information (*Any*)

  The following information types are defined:

  `@CatID` ID of the OFML program that contains the catalog data (*Symbol*).

  This ID can be different from the program ID of the implicit instance if the OFML program of the article instance does not contain catalog data.

  `@CatV` The version of the OFML package of the OFML program with the catalog data (*String*).

[34] in base type *OiPart* as of OI version 1.42.0 from fall 2022
[35] whose implementation is based on the application callback *::ofml::app::getPDLanguage()*

`@ArtNr` The basic article number (*String*) specified in the catalog entry.

This may differ from the basic article number of the implicit instance[36] if an article exchange took place as a result of the assignment of the variant code from the catalog entry (see below).

`@VarCode` The variant code specified in the catalog entry for assigning property values different from the initial (basic) configuration of the article (*String*).

Which of these possible information items are passed in parameter *pInfo* and the order of the items depends on the applications.

The assignment takes place immediately after the commercial initialization of the article instance (see also appendix E).

The default implementation stores the passed information in a corresponding member variable.

- *getCatalogInfo()* → *Any[]* | *Void* **new**

  The function returns information about the entry in the catalog of the application that was used when inserting/creating the article instance.

  If no information has been assigned to the implicit instance (see function *setCatalogInfo()* above), a value of type *Void* is returned. Otherwise the return value is as specificied for parameter *pInfo* of function *setCatalogInfo()*.

  The information can be used by the application to identify the catalog entry that was used when inserting/creating the article instance[37].

  The default implementation returns the current value of the member variable to which the information is assigned during *setCatalogInfo()*.

[36] according to function *getArticleSpec()* in section 2.2)
[37] e.g. to get images that are stored in the catalog for the article

3 Interface *CompositeArticle*

3.1 Synchronization with the basket structure

- *getSubArticleIDs()* → *String[]*

  The function returns a List or Vector containing the IDs for those child objects of the implicit instance that have to be represented in the basket structure, i.e. which represent sub-articles.

  The IDs are created and managed by the composite instance. An ID must uniquely identify a sub-article in the context of the composite instance.

  The ID of a sub-article must not change during the processing of a change of a property of the composite article!

  Possible methods for generating the IDs are:

  - using the OFML type and/or the geometrical position of the child instance
  - using the local name of the child instance
  - using the property setting of the composite instance, that has caused the creation of the child instance[38]

- *getSubArticle(pID(String))* → *MObject* | *Void*

  The function returns the reference to the child instance of the implicit instance, that represents the sub-article with the specified ID.

  The passed ID is expected to come from a previous call of method *getSubArticleIDs()* (see above) of the implicit instance.

  If there is no child instance with the passed ID, a value of the type *Void* is returned.

When creating the (persistent) basket representation, for all sub-articles of the composite article there will be created a corresponding sub item (position) in the basket structure. For the sub items, the respective ID (from *getSubArticleIDs()*) is saved.

Using this, the composite article instance can be completely recreated from the basket representation (see also appendix E).

[38] but consider situations, where a certain property setting may lead to the creation of more than one child instance

A Alphabetical index of functions

B Standard categories

The following categories are predefined for article instances:

`@PseudoArticle`

The instance represents a pseudo-article, i.e. it is an object that does not represent a physical/real article, but merely implements the interface *Article* for technical (and possibly historical) reasons. Therefore, such instances must be handled in a special way during order processing.

If an article instance belongs to this category, a corresponding indicator is set in the persistent basket representation of the article and this is transferred to the order processing via the relevant interfaces and data formats.

Typical use cases for this category are:

- Planning groups and other composite articles that combine other articles from a planning or other point of view.
- Instances intended for planning and other auxiliary purposes (placeholder, dummies).

Typically, the article number of a pseudo article is not available/known in the manufacturer's ERP system, but is created artificially as part of the OFML data creation. In this case, the manufacturer must/should be asked whether the category must be set.

If the article number is known in the manufacturer's ERP system, it possibly is used there also only for auxiliary purposes, e.g. for managing of typicals. Here, the manufacturer has to determine whether the category must be set for proper order processing.

Instances that belong to category *@PseudoArticle* have the following requirements:

1. The instance must also belong to the interface category *@IF_Article*.
2. Method *getArticlePrice()* (section 2.4) must not return a price, i.e. the return value must be of type *Void*. (If this requirement is not met, the category is ignored for a given article instance.)

OFML applications do not display a price or article number for instances of this category, but may display the article short text[39]. Furthermore, articles of this category are excluded from commercial calculation.

See also the following categories *@NonOrderArticle* and *@NonOfferArticle*.

`@NonOrderArticle`

The instance represents a pseudo-article that is not relevant for order processing.

If an article instance belongs to this category, a corresponding indicator is set in the persistent basket representation of the article. Articles with this indicator will not be included in printed orders as well as in documents for electronic order data exchange (e.g. OEX document type ORDERS [oex])[40].

Article instances of this category are assumed to (also) belong to category *@PseudoArticle*. (If this requirement is not met, the category is ignored for a given article instance.)

The determination of this category for a given article must be made in consultation with the manufacturer.

If the category is set for a composite article, its sub-articles will be raised in the order document to the level of the composite article.

[39] if available

[40] The category therefore has no influence on basket views (article lists), that are not explicitly typified as an *order*.

`@NonOfferArticle`

The instance represents a pseudo-article that should not be shown in customer offers.

If an article instance belongs to this category, a corresponding indicator is set in the persistent basket representation of the article. Articles with this indicator will not be included in printed offers as well as in documents for electronic offer data exchange (e.g. OEX document type QUOTES [oex])[41].

Article instances of this category are assumed to (also) belong to category *@PseudoArticle*. (If this requirement is not met, the category is ignored for a given article instance.)

The determination of this category for a given article must be made in consultation with the manufacturer.

If the category is set for a composite article, its sub-articles will be raised in the offer document to the level of the composite article.

`@SALESONLY_ARTICLE`

The instance represents an article without a specific geometry.

Possible special treatments for article instances of this category are:

- Hide when generating the image for parent article instance (if any)
- No dynamic generation of an article image[42]

[41] The category therefore has no influence on basket views (article lists), that are not explicitly typified as an *offer*.
[42] instead use of an image stored in the catalog data (if available)

C OFML variant code

For the structure of the OFML variant code, the provisions apply that are specified for the variant code of the predefined OCD coding scheme `KeyValueList` [ocd], namely:

Every currently valid/visible configurable property is presented in order according to the OCD property table as follows:

`<class>.<property>=<value>`

The semicolon is used as the separator character between the properties.

For currently non-valued optional and restrictable characteristics the value identifier "`VOID`" is used. When coding the values of evaluated properties, there is no padding with blanks according to the specification in the length field of the property table, i.e. only the significant characters are displayed.

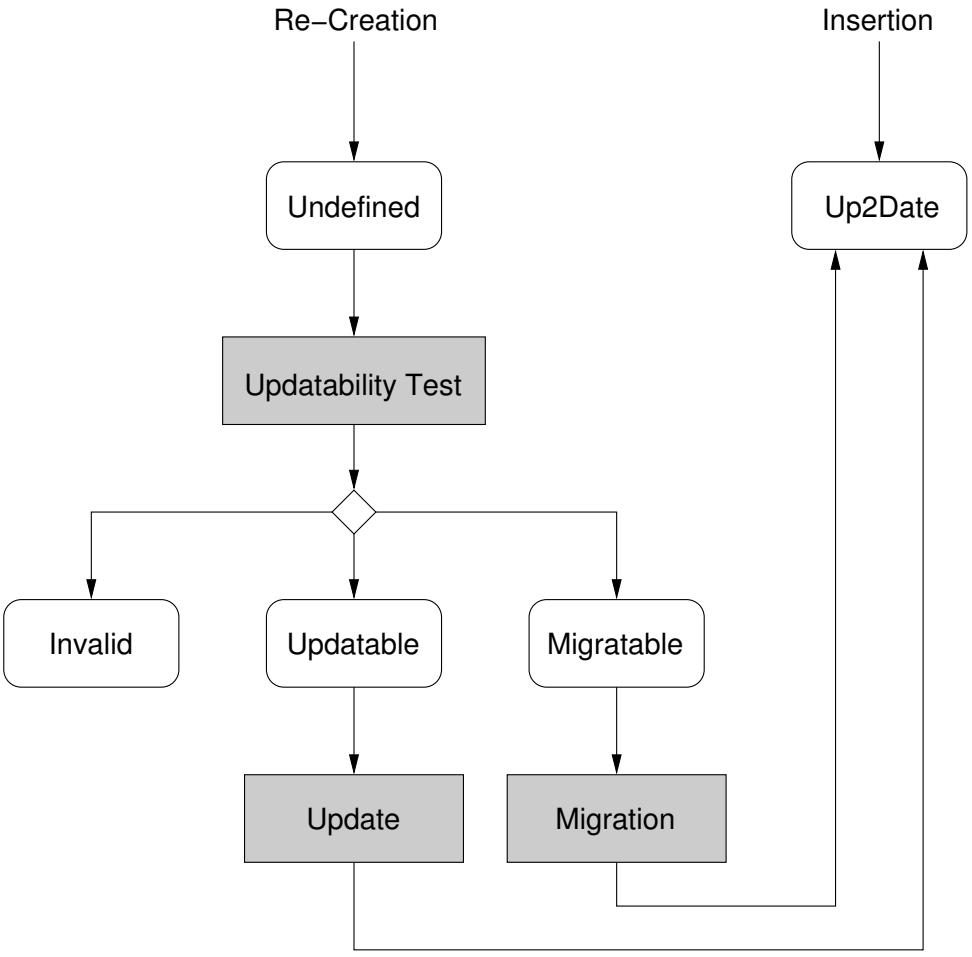
If the manufacturer-independent OFML variant code, as recommended in section 2.6, is used to update saved article configurations, the following issue can occur[43]:

If the OCD data is exported from an ERP resp. PPS system that does not know the concept of a property class, and where arbitrary property classes are generated by the export routine, the update could fail because the properties may not be "recognized" due to a changed property class.

Therefore, OFML applications have to offer and support an alternative updatability check, that ignores property classes in the variant code.

[43] The problem also occurs with manufacturer-specific variant codes, which are generated with the predefined OCD coding scheme `KeyValueList`.

D Update state



E Initialization of article instances

When **inserting** an article **from the catalog** of an OFML application, the following process is expected/assumed regarding the initialization of the article instance:

1. Definition of the program context.

   The OFML program ID is read from the catalog data by the application. The type for the program info object (base type *OiProgInfo*) and the type of the product database (base type *OiProductDB*) are determined from the registration data for the installed OFML package with the program ID.

   With this information the program context is defined by appropriate method calls on the global planning instance, see the functions *setProgram()*, *addInfo()* and *addProductDB()* of base type *OiPlanning* in [ofml].

2. Creation of the article instance.

   The type to be used is determined based on the basic article number read from the catalog data by calling method *article2Class()* on the global planning instance (see base type *OiPlanning* in [ofml])[44].

   This step also includes calling the initialization function *initialize()* for the created instance.

3. Commercial initialization.

   This is done by assigning the basic article number and the (possibly empty or partially determined) manufacturer-specific variant code using calls of methods *setArticleSpec()* and *setXArticleSpec()* on the article instance (see section 2.2).

   The basic article number and the variant code are taken from the catalog data.

4. Other initialization.

   Here, using the OFML interface *Property*, specific property settings can be made for the article instance, e.g. based on user profiles.

   Furthermore, method *setCatalogInfo()* (see section 2.8) is used here to assign information about the the catalog entry that was used when inserting the article.

   Finally, method *articleInserted()* is called for the created article instance on the global planning instance (base type *OiPlanning*).
   Here, an event, e.g. `@ArticleInserted`, can be reported to the global ChangeManger (base type *OiChangeManager*) by the specific subclass of *OiPlanning* used in the respective OFML application.

When **restoring** an article instance based on a saved **basket representation**, the following process is expected/assumed regarding the initialization of the article instance:

1. Definition of the program context.

   As above, but the OFML program ID is taken from the basket representation.

2. Creation of the article instance.

   As above, but the basic article number is taken from the basket representation.

3. Commercial initialization.

   This is done by assigning the basic article number and the manufacturer-independent OFML variant code using a call of method *setupConfiguration()* on the article instance (see section 2.6).

   The basic article number and the variant code are taken from the basket representation.

   If the article instance does not have the update status `@Up2Date` after the method call (see section 2.6), the restoration has failed (and the process will be aborted).

[44] Mapping tables standardized in OFML Part VI (OAM) are used for this.

4. Other initialization.

   Here, the additional state code stored in the basket representation is assigned by calling method *setAddStateCode()* on the article instance (see section 2.3).

   Furthermore, method *setCatalogInfo()* (see section 2.8) is used here to assign information about the the catalog entry that was used when inserting the article. (This assumes that this information is stored in the basket representation.)

5. Initialization of sub-articles.

   In the case of a composite article, for each sub-item (position) in the basket, the *SubArticleID* stored there is read and the instance representing the sub-article with this ID is determined by calling method *getSubArticle()* on the composite article instance (see section 3.1).

   Steps 3 to 5 are repeated for each sub-article instance, where the call of *setCatalogInfo()* (in step 4) is omitted for it.

At the end of step 1 (in both scenarios), the application informs the global planning instance (base type *OiPlanning*) by calling method *setCreationMode(pMode(Int))* which of the two scenarios is present. The corresponding values for the parameter are:

`0` inserting an article from the catalog

`1` restoring an article instance based on a saved basket representation

When implementing functions of the interfaces *Article* and *CompositeArticle*, this mode can be taken into account if required. The mode can be determined by calling the corresponding method *getCreationMode()* on the global planning instance[45].

The mode is particularly relevant for the **creation and initialization of sub-article instances** of composite articles:

`0` In this scenario, the complete creation and initialization of initial sub-article instances takes place during the methods *setArticleSpec()* resp. *setXArticleSpec()* of the composite article instance, i.e., steps 1 to 4 of the scenario are executed by the composite article instance for the initial sub-articles.

`1` Since in this scenario steps 3 and 4 for *all* sub-articles of a composite article are triggered by the application (step 5) after the composite article instance has been created and initialized, for performance reasons no sub-article instances should be created and initialized during *setArticleSpec()* resp. *setXArticleSpec()*[46].
Step 2 – creation of the sub-article instance including positioning – takes place during method *setAddStateCode()* of the composite article instance[47]. For that purpose, the corresponding information[48] for *all* (current) sub-article instances[49] have to be saved in the *AddStateCode* (see method *getAddStateCode()*, section 2.3).

[45] The global planning instance can be determined using the global function *oiGetPlanning()*.
[46] via *setupConfiguration()*
[47] Accordingly, sub-article instances created nevertheless during *setupConfiguration()* are removed here!
[48] at least: SubArticleID, program ID, basic article number, position, rotation
[49] i.e., also for sub-article instances that were added or changed interactively by the user

F Document history

F.1 Version 1.3 vs. Version 1.2 (2024-10-21)

- Clarifications and minor stylistic improvements in the introduction (section 1).
- Additions in appendix E, especially regarding the creation and initialization of sub-article instances.

F.2 Version 1.2 vs. Version 1.1 (2024-01-11)

- Moved function *getArticleObj()* from section 2.4 to section 2.8.
- Description of the possibility of using product data caches for identical article configurations (section 2.4).
- New function *getArticleClassifications()* in section 2.4.
- New appendix B contains specifications of standard categories.

F.3 Version 1.1 vs. Version 1.0 (2022-05-02)

- New function *getPDLanguage()* in section 2.8.