

Formal Introduction into the Concept-Oriented Data Model

Alexandr Savinov

<http://conceptoriented.com>

First created: 20.06.2005

Last update: 04.08.2005

1. General terms

We separate two dual types of element composition: collection and combination.

Collection is a number of elements connected by operation of sum. We denote a collection by enclosing its elements into curly brackets: $C = \{e_1, e_2, \dots, e_n\}$. If an element belongs to a collection then we write it as follows: $e_j \in C, j = 1, 2, \dots, n$. This type of collection is called a physical collection in order to distinguish it from logical collections described below. Elements within a collection are identified by means of their *references*.

Combination is a number of elements connected by operation of product. We denote a combination by enclosing its elements into angle brackets: $O = \langle e_1, e_2, \dots, e_n \rangle$. If an element belongs to a combination then we write it as follows: $e_i \in O, i = 1, 2, \dots, n$. Elements within a combination are identified by means of its *position*.

Positions are known in advance and are used for accessing and manipulating data in queries or programs but they cannot be stored in data. References are not manipulated explicitly and are unknown but they are stored as the contents. (This has some exceptions such as well known references for bootstrapping purposes.) Shortly, positions are elements of *code* (of program, query or other type of code), while references are elements of *data* (of objects, records or other types of data). Another analogy is that in tables columns are identified as positions while rows are identified as references (in great extent because of this duality of rows and columns tables are so successful in data modelling).

Logical collection and logical membership is defined as a dual notion with respect to combinations and membership in combinations. If element C is a member of combination $O = \langle \dots, C, \dots \rangle$ then O is a member of the collection $C: C = \{ \dots, O, \dots \}$. If we draw collection C and combination O as two points then the two types of membership can be shown as two opposite arrows: $\bar{O} \in C$ and $C \in O$, where the former means membership in a logical collection, and the latter means inclusion into a combination. We will follow a convention where combinations are written below their elements and collections are drawn above their elements.

Any element has two parts:

- it is a physical collection of other elements (collectional part), and
- it is a combination of other elements (combinational part).

Thus it can be represented as a pair of collection and combination: $e = \{c_1, c_2, \dots, c_m\} \langle o_1, o_2, \dots, o_n \rangle$. One or both parts can be empty. We will follow a convention that elements with non-empty collectional part will be written from the uppercase letter while elements with non-empty and the most important combinational part will be written from the lowercase letter.

2. Physical Structure and Logical Structure

A two-level concept-oriented model consists of one root element R which physically includes a set of *concept* elements $R = \{C_1, C_2, \dots\}$ where each concept physically includes a set of *item* elements $C_j = \{i_1, i_2, \dots\}$.

Physical inclusion is supposed to be persistent and cannot be changed during life cycle of elements – they are created, manipulated and deleted within one parent physical collection. This property is used for representation purposes. And this is precisely why we additionally assume that the physical structure is has a form of tree: any element is physically included into one parent element. In other words, each item has one parent concept and all concepts have one parent (the root of the model).

The physical structure of the model expresses only the fact of existence of elements and allows us to represent them by storing their references. However, the model needs also a logical structure where elements are connected with one another. If physical part is defined by collectional parts of elements then logical structure is defined by combinational parts of elements. In contrast to physical structure, logical structure is not persistent and can be changed. An element can be logically included into several parent elements simultaneously.

The logical structure is described by representing each element as a combination of other elements from the same level. Logical structure at the first level of concepts is called the model *syntax* and it is described in the next section. The logical structure at the second level of items is called the model *semantics* and it is described in section 4. An arbitrary case of multi-level models is not described here.

3. Syntax

In the previous section we defined a concept at the physical level as a collection of its items. At the logical level a *concept* is a combination of other concepts from the model:

$$C = \langle C_1, C_2, \dots, C_n \rangle$$

Concepts C_1, C_2, \dots, C_n are called *superconcepts* of rank 1. Concept C is called *subconcept* of rank 1.

The concept definition establishes a *subconcept-superconcept relation* between concepts:

$$C_j \in C, \quad j = 1, 2, \dots, n$$

Cycles in this relation are not permitted. (In practice we can easily permit loops with special assumptions.)

If a concept does not have superconcept then it is assumed to be a *top* concept T . If a concept does not have a subconcept then it is assumed to be a bottom concept B . Direct subconcepts of the top concept are called *primitive* concepts. Top concept is a direct or indirect superconcept for all other concepts, i.e., all concepts are logically included into the top concept collection. Bottom concept is direct or indirect subconcept for all other concepts.

If one of two concepts is a direct or indirect parent of another then they are called *syntactically dependent* or parallel concepts; otherwise the two concepts are referred to as *syntactically independent* or orthogonal. In particular, all primitive concepts are syntactically independent.

Each superconcept has a uniquely identified position within a concept definition, which is called *dimension* (of rank 1 or local dimension):

$$C = \langle x_1 : C_1, x_2 : C_2, \dots, x_n : C_n \rangle$$

Here superconcepts C_1, C_2, \dots, C_n are called *domains* for dimensions $x_1, x_2, \dots, x_n : C_j = \text{Dom}(x_j)$. All dimensions x_1, x_2, \dots, x_n are unique and are used for access while some concepts within the combination $\langle C_1, C_2, \dots, C_n \rangle$ can be the same, i.e., dimensions may have one and the same domain. Normally dimensions (concept positions within a combination) are identified by names or integer values.

A dimension of rank k is a sequence of k dimensions of rank 1 separated by dots where each next dimension in the sequence belongs to the domain of the previous one:

$$x^1.x^2.\dots.x^k, \quad \text{where } x^j \in \text{Dom}(x^{j-1}), \quad j = 1, 2, \dots, n$$

Dimensions will be frequently prefixed by the first concept corresponding to the first dimension in the sequence:

$$\text{Dom}(x^0).x^1.x^2.\dots.x^k$$

Also we frequently will use the terms dimension and domain interchangeably.

Primitive dimension has primitive domain (of the last element in the sequence). The number of *all* primitive dimensions of a concept is referred to as the concept *primitive dimensionality*. The primitive dimensionality of the model is that of the bottom concept. The maximal rank of a primitive dimension of a concept is referred to as the *concept rank*. The rank of the model is that of the bottom concept. Thus each model is characterized by its dimensionality (width of multidimensional space) and its rank (depth of the hierarchy).

Dually, each concept is a logical collection of other concepts:

$$C = \{S_1, S_2, \dots, S_n\}$$

Here C is a superconcept of rank 1 and S_1, S_2, \dots, S_n are subconcepts or rank 1.

A *inverse dimension* is a dimension with the opposite direction. We denote inverse dimensions by enclosing the corresponding dimension into curly brackets. If $x^1.x^2.\dots.x^k$ is a dimension of rank k then $\{x^1.x^2.\dots.x^k\}$ is an inverse dimension of the same rank k . In contrast to dimensions which identify superconcepts, the role of inverse

dimensions is dual – they identify subconcepts. The domain of inverse dimension is the very first concept in the sequence, i.e., if $\text{Dom}(x^0).x^1.x^2.\dots.x^k$ is a inverse dimension then $\text{Dom}(x^0)$ is its domain.

The number of inverse dimensions of a concept with the domain in the bottom concept is referred to as the concept inverse dimensionality. The primitive inverse dimensionality of the model is that of the top concept. The model primitive dimensionality is equal to the model primitive inverse dimensionality because each primitive inverse dimension is a primitive dimension with opposite direction.

Such a structure of concepts can be represented by a concept graph where nodes are concepts and edges are instances of the subconcept-superconcept relation $C \bar{\in} C_j$ identified by dimensions. Each path in the concept graph consisting of k edges leads from the source concept to a superconcept of rank k . Such a path is interpreted as a dimension of rank k , which is identified by a sequence of k local dimensions. The model dimensionality and inverse dimensionality is the number of paths from the bottom to the top to the from top to the bottom, respectively. There may be several different paths between a concept and a superconcept. The length of the longest path is the model rank.

4. Semantics

An item is a combination of other items or nulls:

$$i = \langle i_1, i_2, \dots, i_n \rangle$$

Items i_1, i_2, \dots, i_n are called *superitems* of rank 1. Item i is called *subitem* of rank 1. Thus each item is a combination of its superitems and a logical collection of its subitems.

In the presence of *syntactic constraints* each item (physically) belongs to one concept where it is called an instance of this concept. In this case each item can combine only superitems from its superconcepts:

$$i = \langle i_1, i_2, \dots, i_n \rangle, \text{ where } i \in C = \langle C_1, C_2, \dots, C_n \rangle \text{ and } i_j \in C_j$$

Using syntactic constraints we can effectively restrict possible items of a concept.

Here is the definition of the concept-oriented model. Two-level concept-oriented model is defined as the root element physically including a number of concepts:

$$R = \{C_1, C_2, \dots, C_n\},$$

where each concept is a combination of some other concepts (with top and bottom concepts) and physically includes a set of items:

$$C_j = \{i_1, i_2, \dots, i_{m_j}\} \langle C_{j_1}, C_{j_2}, \dots, C_{j_n} \rangle$$

where each item is a combination of some other items taken from the corresponding superconcepts:

$$i_j = \{ \langle i_{j_1}, i_{j_2}, \dots, i_{j_n} \rangle \in C_j, i_{j_k} \in C_{j_k} \}$$

5. Model Definition

Depending on the number of levels in the physical hierarchy we select several types of model:

- one-level model has one root and a number of data items in it,
- two-level model has one root, a number of concepts in it each of them having a number of data items,
- multi-level model has an arbitrary number of levels.

In this paper we consider only two-level concept-oriented model defined as consisting of the following constituents:

[Root] One root element is a physical collection of concepts, $R = \{C_1, C_2, \dots, C_n\}$,

[Syntax] Each concept is (i) a combination of other concepts (called *superconcepts* while this concept is a *subconcept*), $C = \langle C_1, C_2, \dots, C_n \rangle$, and (ii) physical collection of *data items* (or concept instances);

$$C_j = \{i_1, i_2, \dots\},$$

[Semantics] Each data item is (i) a combination of other data items (called *superitems* while this item is a *subitem*), $i = \langle i_1, i_2, \dots, i_n \rangle$, and (ii) empty physical collection,

[Special elements] if a concept has no superconcepts then it is referred to as primitive and its superconcept is one common *top concept*; and if a concept has no subconcepts then it is assumed to be one common *bottom concept*, and if an item does not have one of its superitems then it is assumed to be common *null item*.

[Cycles] Cycles in subconcept-superconcept relation and subitem-superitem relation are not allowed, and

[Syntactic constraint] Each data item from a concept may combine only items from its superconcepts.

5. Access Paths and Derived Properties

Projection. If $I \subseteq C$ is a subset of items from concept C and $x = x^1.x^2.\dots.x^k$ is one of dimensions of concept C then $I \rightarrow x = I \rightarrow x^1.x^2.\dots.x^k$ is referred to as *projection* of I to $\text{Dom}(x)$ along dimension x which is defined as a subset of items from $\text{Dom}(x)$ referenced by items from I via the dimension x .

$$I \rightarrow x = \{u \in U = \text{Dom}(x) \mid i.x = u, \forall i \in I \subseteq C\}$$

It is important that projection does not include superitems more than once. Null is interpreted as absence of superitem. If we need to include all superitems referenced by items from I as many times as they are referenced then we use dot, i.e., $I.x$ returns all superitems referenced from I even if they occur more than once.

Deprojection. If $I \subseteq C$ is a subset of items from concept C and $\{x\} = \{x^1.x^2.\dots.x^k\}$ is one of its inverse dimensions with the domain in subconcept S then $I \rightarrow \{x\} = I \rightarrow \{x^1.x^2.\dots.x^k\}$ is referred to as *deprojection* of I to S along inverse dimension $\{x\}$ which is defined as a subset of items from S referencing items from I via dimension x :

$$I \rightarrow \{x\} = \{s \in S = \text{Dom}(\{x\}) \mid s.x = i, \forall i \in I \subseteq C\}$$

Access path. *Access path* is a sequence of segments P^j , $j = 1, 2, \dots, r$ separated by arrow or dot where each segment P^j is either a dimension or an inverse dimension of the domain of the previous segment. Each segment is applied to all items returned by the previous segment and returns a result collection according to the definition of projections and deprojection.

An access path consists of upward and downward segments in the concept graph. Upward segment corresponds to a dimension leading to a superconcept while downward segment corresponds to an inverse dimension leading to a subconcept. Thus access path can change its direction and has a zigzag form.

An inverse dimension in an access path may have constraints, which restrict a set of subitems. The constraints are specified by a predicate f which returns true or false for each subitem from the domain of the inverse dimension:

$$I \rightarrow \{s : S.x \mid f(s)\} = \{s \in S \mid s.x = i \in I \subseteq C \ \& \ f(s) = \text{true}\}$$

Note that the predicate f itself describes constraints by using dimensions, inverse dimensions or arbitrary access path. Here s is an *instance variable* referencing items from S and x is a *bounding dimension*.

Derived property. The mechanism of access path is a convenience method which provides a simple mechanism for accessing data semantics. It is especially useful for defining derived properties of concepts. A *derived property* is a named definition of a query formulated for a concept (as an access path or using other mechanisms) constrained by the current item when executed. For example, we might define a derived property as follows:

$$C = \langle x_1 : C_1, x_2 : C_2, \dots, x_n : C_n, \text{prop} = \text{this}.x_1 \rightarrow a.b \rightarrow \{S.u.v\} \rightarrow c.d \rangle$$

Here the derived property starts from the first dimension of the current item and returns a set of items according to the access path $x_1 \rightarrow a.b \rightarrow \{S.u.v\} \rightarrow c.d$ which consists of the first upward segment leading to a superitem of rank 2, the second downward segment leading to a set of subitems of rank 2 from the concept S , and the last segment which again moves upward to superitems.

6. Queries

Access paths are a convenience method which makes it easy logical navigation and simple querying. In general case we need more powerful mechanisms for querying described below.

Multidimensional query has the following format:

$$C = \{c_1 \in C_1, c_2 \in C_2, \dots, c_n \in C_n \mid f(c_1, c_2, \dots, c_n)\}$$

Here in the first part before the bar we describe a set of source concepts and their instance variables, which define the universe of discourse of this query $\Omega = C_1 \times C_2 \times \dots \times C_n$ where each possible item is a combination of individual items $\omega = \langle c_1, c_2, \dots, c_n \rangle \in \Omega$. In the second part after the bar symbol we specify constraints imposed on the items from the universe of discourse as a predicate f . The result collection includes items from the universe of discourse for which the predicate is true: $C = \{\omega \in \Omega \mid f(\omega) = \text{true}\}$.

Elements of collection are represented by means of references, which are unique for each new collection. The items from the collection then provide access to the original items. However, frequently we need to include some information by value as new dimension values of items in the collection. For simple cases this can be done by specifying them after the multidimensional query:

$$C = \{c_1 \in C_1, c_2 \in C_2, \dots, c_n \in C_n \mid f(c_1, c_2, \dots, c_n)\} \langle v_1(c_1, c_2, \dots, c_n), \dots \rangle$$

Here $v_1(c_1, c_2, \dots, c_n)$ is a function that returns a single item given instance variables. Note that this function itself may include complex multidimensional queries and access paths applied to instance variables.

It is important to understand that each use of curly brackets evaluates to a absolutely new collection, which is a new subconcept to the source concepts. We can assign this new collection to a new variable or it can be used anonymously. For example, $C = \{c_1 \in C_1, c_2 \in C_2, \dots, c_n \in C_n \mid \dots\}$ defines a new collection with its reference stored in variable C . This new collection is a subconcept to the source collections C_1, C_2, \dots, C_n , which are superconcepts. One important principle for building new collections is that all the source collections have to be already defined and constructed before the new collection (subconcept) can be built.

The source collections can be themselves specified via independent queries rather than using references to existing concepts. For example, we might want to restrict a set of items in some source concepts and then we need to write it as a nested query:

$$C = \{c_1 \in \{C_1 \mid f_1\}, c_2 \in \{C_2 \mid f_2\}, \dots, c_n \in \{C_n \mid f_n\} \mid f(c_1, c_2, \dots, c_n)\} \langle v_1(c_1, c_2, \dots, c_n), \dots \rangle$$

Here we did not show instance variables for nested collections, which might be written as follows:

$$\{i \in C_1 \mid f_1(i)\} \langle v(i), \dots \rangle$$

Thus nested collections are normal collections just like their external collection where they are used. It is important that each nested source collection is evaluated before its external collection.

Nested collections can be also used in the predicate part of the query but here they play a different role than the source nested collections. These collections are evaluated in the context of their external queries (rather than before the external query for source nested collections). In particular, the predicate nested collections can use instance variables from their external contexts. Thus for each collection its instance variables defined for source collections are visible from all internal collections in the predicate part. For example, in some internal collection we might impose constraints by using its own instance variables, its external collection instance variables and even instance variables from the parent query context.

It is important to understand that each new collection in query is different from each of its source collections even if we use only one source collection. This new collection is a new subconcept which has its own independent set of items with their own references. For example, a collection $\{i \in C \mid i.prop = 5\}$ which selects items with the specified property has nothing to do with the source collection C . Its items will have different references and it will be a subconcept to C . In particular, it is important that C will still have the same original set of items in it and whenever we use it again somewhere in queries its will not change.

Sometimes however we want to restrict the number of items in some concept or new collection so that these changes are visible to all other queries. Moreover, we want these constraints were propagated over the available concepts. In this case we need to redefine this concept or collection. This new definition and all its consequences will be then visible in other queries (but not in the external context). For example, if we have a concept C and want to restrict its set of items so that these changes are visible for all other queries then we write it as follows: $C = \{i \in C \mid i.prop = 5\}$.

In this way we can effectively redefine any concept. As a side effect of such a redefinition we have these constraints propagated downward in the lattice. This means that if an item was excluded from the concept then all its subitems are also excluded from their concepts.