RxO system.
Simple semantic approach for representation complex objects data in table form.

Evgeniy Grigoriev (C) 2011

Grigoriev.E @ gmail.com

Context

> Introduction.

Since time of database manifestos [1,2,3] a lot of papers and the systems have been being created with theoretical and practical attempts to unite properties of object-oriented programming languages and relational DBMS in single system. There is no common decision and existing ones are not popular yet in circle of practical developers. The theme is still actual.

Offered approach is based on simple idea to use the same names and their combinations to represent the same values both in form of complex objects and in table form. It has been realized in prototype called "RxO-system".

The RxO-system emulates activity of traditional DBMS server. It executes incoming commands of non-procedural language and outputs the results found. It consists of two parts 1) translator and 2) "programmable table machine" (MS SQL Server 2005 is used) which is executive part of system. The own function of translator part is only translation of income commands in commands of "programmable table machine". The translator does not create any variables for intermediate object representation. Also it uses neither internal iterators nor cursors for any group operations on sets of complex objects. The RxO-system is neither an OODBMS nor an ORM-system.

RxO-system acts as …
1) … the tool for describing of enterprise objects. RxO DDL allows
   - creating of classes with complex (0NF) structure and references between classes,
   - encapsulating state, data persistence property, behavior and constrains in separate object components.
   - redefining these properties during multiple class inheritance
2) … the environment for persistent objects. RxO DML allows creating objects, changing the state of existing objects, method executing. These actions can be run on sets of objects without extents or iterators.
3) … the tool for data querying. RxO DQL allows ad-hoc queries over sets of complex polymorphic objects to get their data (not object themselves)

The prototype executes new language constructions which are similar to ones existing in current SQL versions. New DDL commands "`CREATE CLASS`" and "`ALTER CLASS … REALIZE`" are used for class defining and describing. New DML commands "NEW" and "DESTROY" are used for object creation and termination. For changing of object data usual SQL DML commands INSERT, UPDATE, DELETE, EXEC are used. Now they can work on data described in terms of complex structures. DQL command SELECT has the same possibilities.

It will be clear below, that the realized approach does not impede usage of traditional table structures and makes possible evolutionary development existing SQL versions in a different way that offered by current SQL standards and existing object-relational DBMSs.

Let's specify that the main aim of this paper is to demonstrate the logic of object and table integration in single system, single language and single namespace. Offered language constructions show only provisional direction, and are not a point of issue.

The model of trade company is used as illustrating example. We try to use meaningful names and comments. Full code of the example will be given in real sequence of commands to demonstrate the simplicity of model creation and usage. Code of example model is concluded in borders.

> Class creation.

Creation of classes includes two separate steps

1) The class specification (command CREATE CLASS …) defines class existence and its interface as a set of valued components and methods. Other classes can be referred as parent ones.
2) For each component and method listed in class specification the realization (command ALTER CLASS … REALIZE …) is set. Each component can be realized as stored or as calculated. Objects of a class can be created after all components of a class are realized.

The specification completely describes the interface which it used to manipulate objects of the class. Parent classes, valued components (i.e. having a value), methods allowing changing objects state and keys are listed in the specification.

Valued components have one of next types.

1) Scalar base type. In the prototype next base types are realized: - BOOLEAN, INTEGER, FLOAT, STRING, DATETIME.
2) Scalar reference types. The name of class is used to define a variable referred to object of this class
3) Sets (SET OF construction). The component of SET type is a set of tuples defined on scalar (both base and reference) types. Keys can be defined for the sets.

Keys also can be defined for classes as a whole. Relationships between classes can be set both by references and by foreign keys.

Our model contains next classes

```
CREATE CLASS BANKS   //Banks - Very simple structure
{
  Name STRING;
  BIC STRING;        // bank ID
}KEY uniqBIC (BIC)   // class key - banks are unique with their ID
;
```

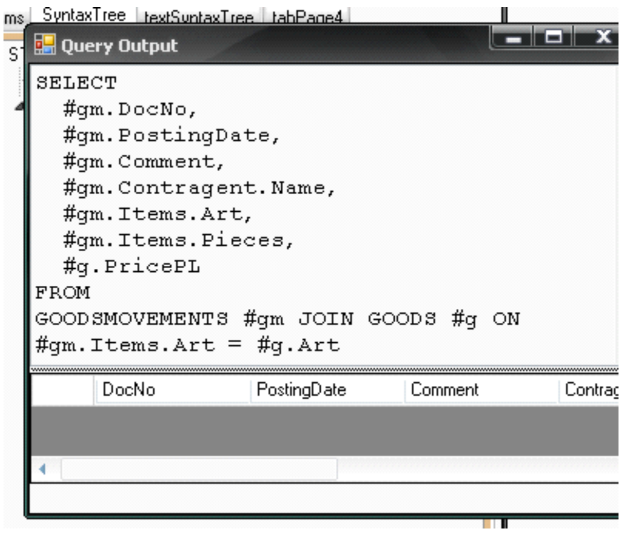
```
CREATE CLASS CONTRACTORS // simple structure too
{
  Name STRING;
  Bank BANKS;             // reference to BANKS
  BankAcc STRING;
  INN STRING;             // fiscal ID
}KEY uniqINN (INN)       // class key – fiscal ID is unique
;
```

```
CREATE CLASS GOODS
{
  Art STRING;        // goods ID (Article)
  PricePL FLOAT;     // default pricelist price
  Turnover SET OF    // all operations on the goods
  {
    Comment STRING;           // operation comment
    Date DATETIME;            // operation date
    DocNo STRING;             // Document number
    Contractor CONTRACTORS;  // reference to CONTRACTORS
    Pieces INTEGER;           // quantity
  }KEY Key4GoodsList (DocNo, Contractor);  //set key
  Pieces2IN INTEGER;  // quantity planned to purchase
  Pieces INTEGER;     // stock quantity
  PiecesRes INTEGER;  // reserved quantity
  PiecesFree INTEGER; // quantity not reserved (free-to-ship)
}KEY UniqArt (Art)   //Class key – Art is unique
;
```

```
CREATE CLASS GOODSMOVEMENTS   // goods operation
{
  DocNo STRING;
  Contractor CONTRACTORS;      // reference to CONTRACTORS
  DocDate DATETIME;            // date of document
  Comment STRING;
  PostIt(inDate DATETIME);     // method used to post the goodsmovement
  PostingDate DATETIME;

  Items SET OF                 // list of goods to be operated
  {
    Art STRING;            //what goods (used in foreign key – see below)
    Pieces INTEGER;       // quantity to be operated
  }KEY uniqArt (Art);          // SET key
}KEY uniqDocNo (DocNo)         // class key
REFERENCE ToUniqArt            // foreign key referencing to class key
Items.(Art) ON GOODS.UniqArt      // only existing Art can be operated
;
```

Class specification is only way to interact with objects of the class. For example as soon as specification are defined next ad-hoc query on specified classes can be executed (its detailed description is later)



```
SELECT
  #gm.DocNo,
  #gm.PostingDate,
  #gm.Comment,
  #gm.Contragent.Name,
  #gm.Items.Art,
  #gm.Items.Pieces,
  #g.PricePL
FROM
GOODSMOVEMENTS #gm JOIN GOODS #g ON
#gm.Items.Art = #g.Art
```

This query will return nothing in this moment. Classes are empty still; their realizations haven't set yet so objects cannot be created. But query itself will be executed successfully by RxO-system because queries use class specifications only to be executed.

Realization is defined separately for each class component with command

```
ALTER CLASS class_name
REALIZE component_or_method_signature
AS...;
```

Valued components can be realized in two ways.

1) As stored ones

```
... AS STORED;
```

Values of such components persist in system (like values are stored in SQL tables). The components containing key fields are realized only as stored in prototype.

2) As calculated by the procedure returning value

```
... AS
{
    procedure_body
};
```

These components act as functions returning scalar or set value.

Methods are realized only as procedures with no returns.

Component realizations are encapsulated in classes. Realization of inherited components and methods can be redefined in child classes (see "Multiply inheritance and polymorphism").

Component realization can be changed in existing nonempty classes.

In our model all components of classes `BANKS` and `CONTRACTORS` are realized as stored

```
ALTER CLASS BANKS
REALIZE
  Name STRING
AS STORED;
```

```
ALTER CLASS BANKS
REALIZE
  BIC STRING
AS STORED;
```

```
ALTER CLASS CONTRACTORS// a number of scalar components can be realized
REALIZE (              // as stored in one command
  Name STRING,
  Bank BANKS,
  BankAcc STRING,
  INN STRING
)AS STORED;
```

Some components of class `GOODS` are realized as calculated. Component `Turnover` is calculated on data of class `GOODSMOVEMENTS` using SELECT expression (these expressions will be described further).

```
ALTER CLASS GOODS
REALIZE
  Turnover SET OF
  {
    Comment STRING;
    Date DATETIME;
    DocNo STRING;
    Contractor CONTRACTORS;
    Pieces INTEGER;
  }KEY Key4GoodsList (DocNo, Contractor)
AS
{
RETURN
SELECT
  #g.Comment,
  #g.DocDate,
  #g.DocNo,
  #g.Contractor,
  SUM(#g.Items.Pieces) AS Pieces
FROM GOODSMOVEMENTS #g   // from GOODSMOVEMENTS
WHERE #g.Items.Art = Art // local in GOODS component Art is used as criterion
GROUP BY
  #g.Comment,
  #g.DocDate,
  #g.DocNo,
  #g.Contractor ;
};
```

The scalar components containing remains and planned quantities are calculated using aggregate queries to class `GOODSMOVEMENTS`.

```
ALTER CLASS GOODS
REALIZE
Pieces2IN INTEGER AS        // quantity planned to be purhcased
{
RETURN
SELECT SUM(#g.Items.Pieces)
FROM GOODSMOVEMENTS #g
WHERE #g.PostingDate IS NULL  // not posted goods operation
  AND #g.Items.Art = Art  // local Art is used as criterion
  AND #g.Items.Pieces > 0;    // positive quantities only
};
```

```
ALTER CLASS GOODS
REALIZE
Pieces INTEGER AS // real stock quantity
{
RETURN
SELECT SUM(#g.Items.Pieces)
FROM GOODSMOVEMENTS #g
WHERE #g.Items.Art = Art
  AND #g.PostingDate IS NOT NULL;  //posted operation only
};
```

```
ALTER CLASS GOODS
REALIZE
PiecesRes INTEGER AS //quantities reserved to ship out
{
RETURN
SELECT SUM(-#g.Items.Pieces) // the value will be presented as positive one
FROM GOODSMOVEMENTS #g
WHERE #g.Items.Art = Art
  AND #g.PostingDate IS NULL
  AND #g.Items.Pieces < 0;   // negative values only is selected
};
```

The component containing free-to-ship quantity is realized with a method used scalar components.

```
ALTER CLASS GOODS
REALIZE
PiecesFree INTEGER AS  //quantity free to ship
{
  DECLARE         //procedure local variables
  {
    tmpPieces INTEGER;
    tmpPiecesRes INTEGER;
  }
  tmpPieces := Pieces;
  IF( tmpPieces IS NULL) THEN tmpPieces := 0;//NULL is replaced with 0
  tmpPiecesRes := PiecesRes;
  IF( tmpPiecesRes IS NULL) THEN tmpPiecesRes := 0;   //NULL -> 0
  RETURN tmpPieces - tmpPiecesRes;  //returns free pieces
};
```

Class `GOODS` contains both calculated and stored components.

```
ALTER CLASS GOODS
REALIZE (
  Art STRING,
  PricePL FLOAT
)AS
STORED;
```

All valued components of class `GOODSMOVEMENT` are realized as stored.

```
ALTER CLASS GOODSMOVEMENTS
REALIZE (
  DocNo STRING,
  Contractor CONTRACTORS,
  DocDate DATETIME,
  PostingDate DATETIME,
  Comment STRING
)AS
STORED;
```

```
ALTER CLASS GOODSMOVEMENTS
REALIZE
  Items SET OF
  {
    Art STRING;
    Pieces INTEGER;
  }KEY uniqArt (Art)
AS
STORED; // the component is persist now
```

The persistence property of component `Items` will be redefined further (see "Multiply inheritance and polymorphism").

Realization of method `PostIt` sets the goods operation posting date and marks the comment

```
ALTER CLASS GOODSMOVEMENTS
REALIZE PostIt(inDate DATETIME)
AS
{
IF(PostingDate IS NULL) THEN
  {
  PostingDate := inDate;              // set the posting date
  Comment := "DONE " + DocNo;         // change comment
  IF( DocDate < '2010.01.01') THEN    // and for old operation
    Comment := "OLD! " + Comment;     // make special mark in comment
  }
};
```

> From paths to table views.

The ability to use paths in DML and DQL commands is the main one. Here the path is name sequence determined by structures and references defined in class specification. Dot notation is used to write a path. Idea of paths (to reflect hierarchy, to specify a part of the general etc) seems obvious enough. The same constructions with the same meaning are used widely in program languages (inc. OO languages).

The paths in RxO-system can be considered as general case of such constructions as they begin with any expression which means set of objects. It can be a name of a class (that means all existing objects of a class), a reference variable name (referred to just single object) or a path ended with reference (that means a subset of objects of a class).

The path including just one name is simplest one.

The type of a path is defined by type of its last element.

In a global context any paths begin with a name of a class (because only these names are defined in global context). The path containing single name of class has reference type. F.e. in our model in a global context next paths exist among other

`BANKS.Name` ${}^{2}_{B}$
`CONTRACTORS` ${}^{1}_{R}$
`GOODSMOVEMENTS.Contractor.Name` ${}^{3}_{B}$
`GOODS.Turnover` ${}^{2}_{S}$
`GOODS.Turnover.Pieces` ${}^{3}_{B}$
`GOODS.Turnover.Contractor.Bank` ${}^{4}_{R}$
etc
(In-line indexes contain short path characteristics: upper one contains path length, lower one defines type of path where "B" means base type, "S" means set type, "R" means reference type, "M" means method.)

In class contexts and in local contexts of method (this-contexts) paths can begin also with name of valued component or local variable if they are references or sets.

In our model next path is defined, among other, in context of class `GOODSMOVEMENT`
`Contractor.Bank.Name` $^{3}_{B}$
In context of class GOODS among other next two paths are defined
`Turnover` $^{1}_{S}$
`Turnover.Conragent` $^{2}_{R}$

The paths of reference or set types always have continuations (post-paths). Such paths will be named as non-terminal ones further. Non-terminal path can have a lot of post-paths. In out model the path `GOODSMOVEMENTS.Contractor` (of reference type `CONTRACTOR`) has, among other, next post-paths (their short characteristics start with sign "+" in the upper index)
`.Name` $^{+1}_{B}$
`.INN` $^{+1}_{B}$
`.Bank` $^{+1}_{R}$
`.Bank.Name` $^{+2}_{B}$
Post-paths begin with a point. Such syntactical technique allows separating the post-paths of some path used in a command from other names possible in the context.

The paths of base types are terminal and have no post-paths.

Any existing in path name of a class or reference can be added with object selection expression.
*...name_of_class_or_reference***[***condition_list***]...**

Here *condition* (analogues of WHERE condition of usual SQL SELECT expression) are applied to scalar post-paths of *class_or_reference*. Conditions can be combined with traditional AND and OR operation. Also new operation is added which runs on attributes of SET components only. It is named as interrows_AND and is written down as simple comma ",". It has low priority. For example, next path can be used to get the reference to the object of class GOODSMOVENTS containing rows both for article "`tShirt01`" and for article "`Hat03`"

```
GOODSMOVEMENTS[.Items. Art = "tShirt01", .Items. Art = "Hat03"]
```

Object selection expression can be nested and combined arbitrarily. For example, next path is correct

```
GOODS[
      .Art BETWEEN …,
      .Turnover. Contractor [.Bank. BIC = …]]
.Turnover
.Contractor[.Name LIKE Name]
```

(Here in last line the value of post-path `.Name` of path `GOODS.Turnover. Contractor` and the value of variable `Name` which should be defined in a this-context are compared).

The base principle of path usage in DML and DQL commands is the next: any valid non-terminal path can be treated as a name of existing table view, some post-paths of this path can be treated as names of attributes of this view .

For example, expression "`GOODSMOVEMENTS [.Date> = ' 2010.01.01 '] .Contractor`" determines set of contractors for which after January, 01st, 2010 goods operations were done. This path has, among other, post-paths "`.Name`", "`.INN`" and "`.Bank.Name`". It means, that in our model the next table view, among other, can be used

```
GOODSMOVEMENTS [.Date> = ' 2010.01.01 '] .Contractor      \\ name of view
      (.Name, .INN, .Bank. Name)                          \\ attributes of view
```

Table view name (i.e. path) and its attributes (i.e. used post-paths) together forms the full signature of this view.

F.e. next table views are accessible, among many others, in our model

```
BANKS
      ( .Name , .BIC)

BANKS
      ( .Name)

GOODS
      ( .Art , .Turnover.DocNo , .Turnover.Date, .Turnover.Pieces )

GOODS.Turnover
      ( .Contractor.Name, .DocNo , .Date, .Pieces )

GOODS[.Art LIKE "Hat%"].Turnover
      ( .Contractor.Name, .DocNo , .Date, .Pieces )
```

Such views will be named as O-views.

It should be understood, that when O-view are being discussed the expression like "`GOODS.Turnover`" is considered only as string name of view which differs from other string name (f.e." `GOODS [.Art LIKE " Hat % "]. Turnover` "). Also expressions "`.Art`", "`.Turnover.DocNo`", "`.Turnover. Date`" are considered as different names of attributes of the view. Thus, in table data representation the same names and name sequences are used which have been set with class specifications. All such expressions *keep the meanings* of both single names and their sequences which were set for classes. Used to name table structures they continue *to mean* complex structures.

It allows turning the data representation from classes with complex hierarchical structures to table O-views in single namespace. Moreover, system turns the data representation imperceptibly for users.

In this way RxO-system simultaneously represents all data both as set of objects of different classes and as set of table O-views. There is no certain relationship between these two sets. One class data can be shown in many O-views. One O-view can show data of different classes (combined by references and\or by inheritance). Set of classes and set of O-views are linked semantically only by common names.

Of course, O-views are not defined strictly and manifestly (like it performs for usual SQL views) in moment when classes are being specified. O-views *can be calculated*. Any command which manipulates data contains anyhow some path and its post-paths which together form signature of O-view. The RxO-system analyzes the found signature to check if it satisfies to the base principle and then creates (calculates) a view to execute a command.

During this calculation the components mentioned in the signature bind realizations of these components (late binding). Because of the RxO-system supposes inheritance of classes and redefinition of realizations during inheritance, each of components can simultaneously realized in different ways for different objects. So, table O-views are polymorphic (see further "Multiply inheritance and polymorphism").

As soon as data are represented in table views, they can be manipulated by usual SQL commands.

> Object creation and existence.

New objects are created by NEW expression which can be used both as separate command and as RValue part of assignment

```
NEW class_name [WITH...]
refVar := NEW class_name [WITH...]
```

Optional part `[WITH...]` contains constructing code which allows object component initialization during creation. This ability is realized for scalar component only in prototype.

```
NEW BANKS WITH SET // new object
    .Name := "CitttiBank",  // component initialization
    .BIC := "999999999999999";
```

There is no necessity to store reference to created object. It can be accessed using group operation on the class. Lost objects are not possible in system. All created objects persist until they will be terminated by command

`DESTROY` $path_R$

Here $path_R$ means that any path of reference type can be used as parameter f.e.

```
DESTROY BANKS[.BIC IS NULL]
```

Objects referenced from other parts of system cannot be destroyed.

Single object reference can be obtained by

`ONE OF` $path_R$

```
NEW CONTRACTORS
WITH SET
  .Name := "Levis",
  .Bank := (ONE OF BANKS [.Name = "CitttiBank"]), //get the reference on
  .BankAcc := "40601000000000000001",             //        existing object
  .INN := "77777777777";
```

Expression `NEW` can be nested

```
NEW CONTRACTORS
WITH SET
  .Name := "X3 retail group",
  .Bank :=              // initialize component with the reference
  (NEW BANKS WITH SET    //    on new object created by nested NEW
    .Name := "VoTaBe25",
    .BIC := "888888888888888"),
  .BankAcc := "40601000000000000333",
  .INN := "770000077777";
```

In our model three objects of class `GOODS` are created to describe some goods.

```
NEW GOODS WITH SET .Art := "tShirt01", .PricePL := 1;
NEW GOODS WITH SET .Art := "Trousers05", .PricePL := 5;
NEW GOODS WITH SET .Art := "Hat03", .PricePL := 3;
```

> Data manipulation.

Object values can be changed with usual SQL commands `UPDATE`, `INSERT` and `DELETE` which use paths as parameters. Command `UPDATE` can be applied both to reference and to set components (`R+S` in the lower index).

`UPDATE` $path_{R+S}$ `SET {` $path^{+1}_{B+R}$`: =...} (, n)...`

Here `path`$^{+1}_{B+R}$ means that post-path used as assignment targets is just single name of field which has base or reference types.

Commands `INSERT` and `DELETE` are applied only to sets.

```
INSERT INTO pathS ({path+1B+R} (, n)) (VALUES ...| SELECT ...)
DELETE FROM pathS
```

F.e. command

```
INSERT INTO GOODSMOVEMENTS[.DocNo = "SomeNo"].Items (.Art, .Pieces)
VALUES("tShirt01", 10);
```

will add specified values as new record of set component `.Items` of `GOODSMOVEMENTS` class objects where value of `DocNo` is `"SomeNo"`. Because of `DocNo` field is the key of class this operation will be performed for one object only, if such object exists at all.

All the commands can be run both in global context and in method bodies. They are applied both to classes and to this-defined components and local variables.

Also simple assignment is possible for this-defined scalar components and local variables in method body. The assignment is applied to any length path which doesn't contains name of SET components

```
localreferencepathB+R := ...
```

Class methods are executed by command

```
EXEC pathM ([parameters](,))
```

In our model two objects of class `GOODSMOVEMENT` are created to describe some goods operation. Records are added into them about operated goods.

```
NEW GOODSMOVEMENTS WITH SET
  .DocNo := "InMove01",
  .Contractor := ONE OF CONTRACTORS[.Name = "Levis"],
  .DocDate :='2010.01.01',                               //old date
  .Comment := " 1st SPL";
INSERT INTO GOODSMOVEMENTS[.DocNo = "InMove01"].Items (.Art, .Pieces)
VALUES("tShirt01", 10);
INSERT INTO GOODSMOVEMENTS[.DocNo = "InMove01"].Items (.Art, .Pieces)
VALUES("Hat03", 10);
INSERT INTO GOODSMOVEMENTS[.DocNo = "InMove01"].Items (.Art, .Pieces)
VALUES("Trousers05", 10);
```

```
NEW GOODSMOVEMENTS WITH SET
  .DocNo := "InMove02",
  .DocDate :='2011.02.02',                                //new date
  .Contractor := ONE OF CONTRACTORS[.Name = "Levis"];
INSERT INTO GOODSMOVEMENTS[.DocNo = "InMove02"].Items (.Art, .Pieces)
VALUES("tShirt01", 20);
```

> Ad-hoc queries.

Like DML commands reviewed below, SELECT-expressions is very similar to usual SQL ones. Any non-terminal paths as names of table sources in FROM part and any their scalar post-paths as names of attributes can be used.

Next query was already executed in the system (see "Class creation")

```
SELECT
  #gm.DocNo,
  #gm.PostingDate,
  #gm.Comment,
  #gm.Contractor.Name,
  #gm.Items.Art,
  #gm.Items.Pieces,
  #g.PricePL
FROM
  GOODSMOVEMENTS #gm
JOIN
  GOODS #g
ON #gm.Items.Art = #g.Art;
```

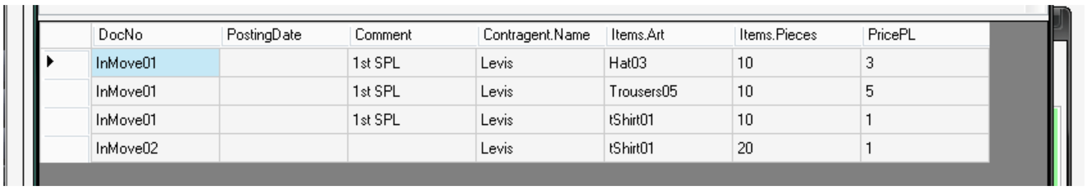
The query (below Q#1) uses two O-views with next signatures

```
GOODSMOVEMENTS
  (.DocNo, .PostingDate, #gm.Comment, .Contractor.Name,
   .Items.Art, .Items.Pieces)
```

and

```
GOODS
  (.Art, .PricePL).
```

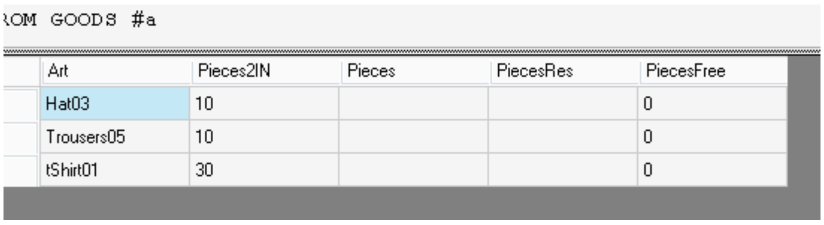
The signatures are correct, meet the base principle and thereby can be calculated with system. Then they can be used by operations usual for table structures. Here JOIN is used with next result.



DocNo	PostingDate	Comment	Contragent.Name	Items.Art	Items.Pieces	PricePL
InMove01		1st SPL	Levis	Hat03	10	3
InMove01		1st SPL	Levis	Trousers05	10	5
InMove01		1st SPL	Levis	tShirt01	10	1
InMove02			Levis	tShirt01	20	1

Below are some other ad-hoc queries and their results

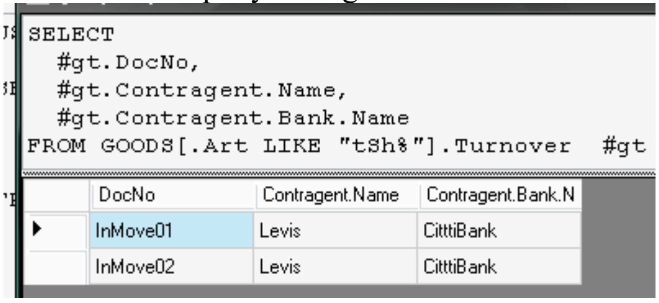
```
SELECT
  #a.Art,
  #a.Pieces2IN,  // pieces to be received
  #a.Pieces,     // pieces on stock
  #a.PiecesRes,  // pieces reserved to be shipped
  #a.PiecesFree  // pieces free to ship
FROM GOODS #a
```



```
ROM GOODS #a
```

Art	Pieces2IN	Pieces	PiecesRes	PiecesFree
Hat03	10			0
Trousers05	10			0
tShirt01	30			0

This is ad-hoc query through multireference structure

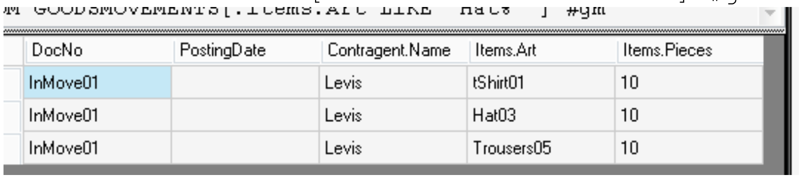


```
SELECT
  #gt.DocNo,
  #gt.Contragent.Name,
  #gt.Contragent.Bank.Name
FROM GOODS[.Art LIKE "tSh%"].Turnover  #gt
```

DocNo	Contragent.Name	Contragent.Bank.N
InMove01	Levis	CitttiBank
InMove02	Levis	CitttiBank

Below condition are applied to determine subset of objects which O-view is built on

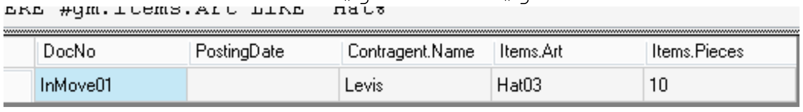
```
SELECT
  #gm.DocNo,
  #gm.PostingDate,
  #gm.Contractor.Name,
  #gm.Items.Art,
  #gm.Items.Pieces
FROM GOODSMOVEMENTS[.Items.Art LIKE "Hat%"] #gm
```



DocNo	PostingDate	Contragent.Name	Items.Art	Items.Pieces
InMove01		Levis	tShirt01	10
InMove01		Levis	Hat03	10
InMove01		Levis	Trousers05	10

Now the same condition is applied to rows when O-view is already built

```
SELECT
  #gm.DocNo,
  #gm.PostingDate,
  #gm.Contractor.Name,
  #gm.Items.Art,
  #gm.Items.Pieces
FROM GOODSMOVEMENTS  #gm WHERE #gm.Items.Art LIKE "Hat%"
```



DocNo	PostingDate	Contragent.Name	Items.Art	Items.Pieces
InMove01		Levis	Hat03	10

> Multiply inheritance and polymorphism.

The specification of child class is union of sets of inherited specifications and own components and methods. Realizations of inherited components and methods can be redefined in child class. For components and methods the late binding is used to bind specification and all realizations what makes these components and methods the polymorphic ones. It allows easy evolution of enterprise model

To illustrate described abilities the new class containing records about value operations is added in our model.

```
CREATE CLASS VALUERECORDS
{
  VRDocNo STRING;       // financial document ID
  VRDate DATETIME;      // record date
  VRComment STRING;     // comment
  ExpectedAmount FLOAT;// expected value amount
  ValueAmount FLOAT;    // recorded value amount
};
```

Class `SALES` containing the information on sales is created. As real sales unite goods and value operations, new class is declare as child one for both `GOODSMOVEMENTS` and `VALUERECORDS` classes.

```
CREATE CLASS SALES
EXTENDED GOODSMOVEMENTS, VALUERECORDS //parent classes
{
  SaleItems SET OF   //SET component: set of sold goods data
  {
    Art STRING;
    Price FLOAT;
    Pieces INTEGER;
  }KEY uniqArtPrice (Art,Price);
}
REFERENCE ToUniqArt SaleItems.(Art) ON GOODS.UniqArt;//only exisding goods
                                                      //can be sold
```

Defined in class `VALUERECORDS` components don't have realizations yet. Let's create them in child class `SALES`.

```
ALTER CLASS SALES
REALIZE
VRDocNo STRING   // as a financial document ID... (inherited component)
AS
{
  RETURN DocNo;  //...goods operaton ID is used
};
```

```
ALTER CLASS SALES
REALIZE
  VRComment STRING // financial document comment ... (inherited component)
AS
{
  RETURN           //contains goods doc ID
   " FinDoc#" + DocNo + "(" + Comment + ")";
};
```

```
ALTER CLASS SALES
REALIZE
  VRDate DATETIME //date of financial operation (inherited component)
AS
{
  RETURN PostingDate; //...is the posting date of goods operation
};
```

```
ALTER CLASS SALES
REALIZE
  ValueAmount FLOAT // recorder value amound ... (inherited component)
AS
STORED              // ...is stored in system
;
```

```
ALTER CLASS SALES
REALIZE
  ExpectedAmount FLOAT //estimated expected amounts ... (inherited component)
AS
{                      //are calculated  ...
  RETURN
    SELECT SUM (-#pi.Pieces*#pi.Price)  //using aggregation query
    FROM SaleItems #pi;
};
```

Component `SaleItems` of child class `SALES` is realized as stored

```
ALTER CLASS SALES
REALIZE
  SaleItems SET OF (own component)
  {
    Art STRING;
    Price FLOAT;
    Pieces INTEGER;
  }KEY uniqArtPrice (Art,Price)
AS STORED;
```

The rule exists in enterprise that only sold quantities of goods have to be shipped. It means that value of component `Items` of parent class `GOODSMOVEMENTS` depends on value of component `SaleItems` of child class `SALES`. To reflect this rule the inherited component `Items` is re-realized as calculated.

```
ALTER CLASS SALES
REALIZE
  Items SET OF //inherited component - new realization
  {
    Art STRING;
    Pieces INTEGER;
  }KEY uniqArt (Art)
AS
{
  RETURN
  SELECT
    #pi.Art,
    SUM(-#pi.Pieces)
  FROM SaleItems #pi
  GROUP BY #pi.Art;
};
```

Method `DoPost` is re-realized too to perform actions typical both for `GOODSMOVEMENTS` and for `VALUERECORDS` classes.

```
ALTER CLASS SALES
REALIZE
  PostIt(inDate DATETIME) //inherited method - new realisation
AS
{
IF(PostingDate IS NULL) THEN
  {
  Comment := "Sale is POSTED! " + Comment; //change good operation comment
  PostingDate := inDate;                   //posting date is set
  ValueAmount :=                           //value amount is recorded
    SELECT SUM (-#pi.Pieces*#pi.Price)
    FROM SaleItems #pi;
  }
};
```

Now class `SALES` is realized fully and two objects of this class can be created and filled with data.

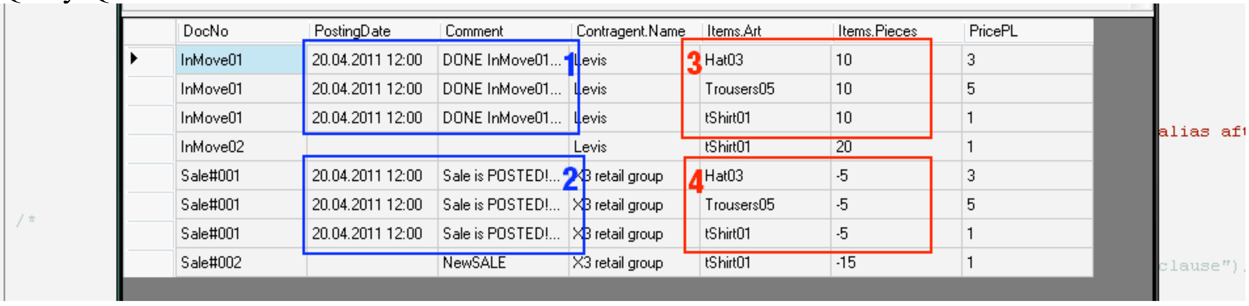
```
NEW SALES WITH SET
  .DocNo := "Sale#001",
  .Comment := "OldSALE",
  .Contractor := ONE OF CONTRACTORS[.Name = "X3 retail group"],
  .DocDate :='2010.11.11'  //old sale
;
INSERT INTO SALES[.DocNo = "Sale#001"].SaleItems  (.Art, .Pieces, .Price)
VALUES("tShirt01", 5, 0.8);
INSERT INTO SALES[.DocNo = "Sale#001"].SaleItems  (.Art, .Pieces, .Price)
VALUES("Hat03", 5, 2.5);
INSERT INTO SALES[.DocNo = "Sale#001"].SaleItems  (.Art, .Pieces, .Price)
VALUES("Trousers05", 5, 4);
```

```
NEW SALES WITH SET
  .DocNo := "Sale#002",
  .Comment := "NewSALE",
  .Contractor := ONE OF CONTRACTORS[.Name = "X3 retail group"],
  .DocDate :='2011.04.04'  //new sale
;
INSERT INTO SALES[.DocNo = "Sale#002"].SaleItems  (.Art, .Pieces, .Price)
VALUES("tShirt01", 15, 4);
```

Now there are two objects of class `GOODSMOVEMENTS` and two objects of child class `SALES` in our model. Let's execute polymorphic method `DoPost` for old events only.

```
EXEC GOODSMOVEMENTS[.DocDate<'2011.01.01'].PostIt('2011.04.20');
```

Query Q#1 returns the next result now



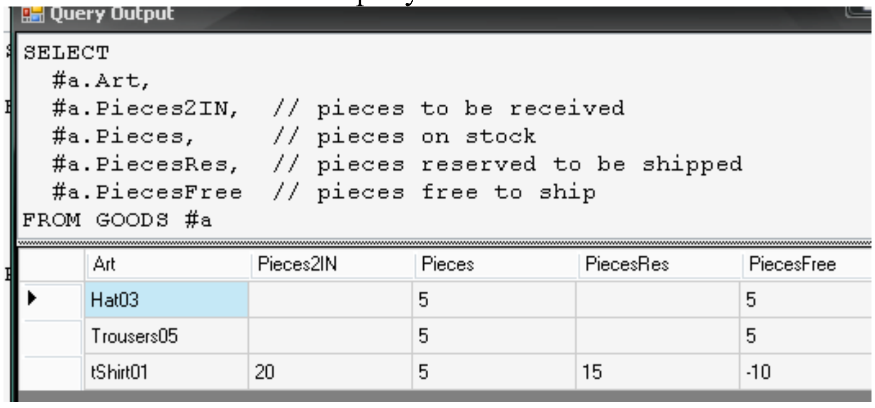
	DocNo	PostingDate	Comment	Contragent.Name	Items.Art	Items.Pieces	PricePL
▶	InMove01	20.04.2011 12:00	DONE InMove01...	Levis	Hat03	10	3
	InMove01	20.04.2011 12:00	DONE InMove01...	Levis	Trousers05	10	5
	InMove01	20.04.2011 12:00	DONE InMove01...	Levis	tShirt01	10	1
	InMove02			Levis	tShirt01	20	1
	Sale#001	20.04.2011 12:00	Sale is POSTED!...	X3 retail group	Hat03	-5	3
	Sale#001	20.04.2011 12:00	Sale is POSTED!...	X3 retail group	Trousers05	-5	5
	Sale#001	20.04.2011 12:00	Sale is POSTED!...	X3 retail group	tShirt01	-5	1
	Sale#002		NewSALE	X3 retail group	tShirt01	-15	1

Columns `PostingDate` and `Comment`, representing corresponding scalar components of class `GOODSMOVEMENT`, contain the values which are result of execution of polymorphic method having two realizations. There are results of execution of realizations both for class `GOODSMOVEMENT` (area 1) and for class `SALES` (area 2). The components `PostingDate` and `Comment` themselves did not change realization at inheritance and are stored.

Other type of polymorphism is presented in columns `Items.Art` and `Items.Pieces`, representing scalar attributes of set component `Items` of class `GOODSMOVEMENT`. Here the component itself has changed the realization. In area 3 there are values how they are stored in system. Area 4 contains results of aggregate query from the data stored a component `SaleItems` existing only in child class `SALES` as it realized in this class.

Query Q#1 has not changed since the moment of first use, when class `GOODSMOVEMENT` was only specified and class `SALES` did not exist at all. But now data represented in this query are the result of complex and heterogeneous calculations over the set of objects of both these classes. It is reached by usual OO-mechanisms of inheritance and the polymorphism realized both for methods and for valued components.

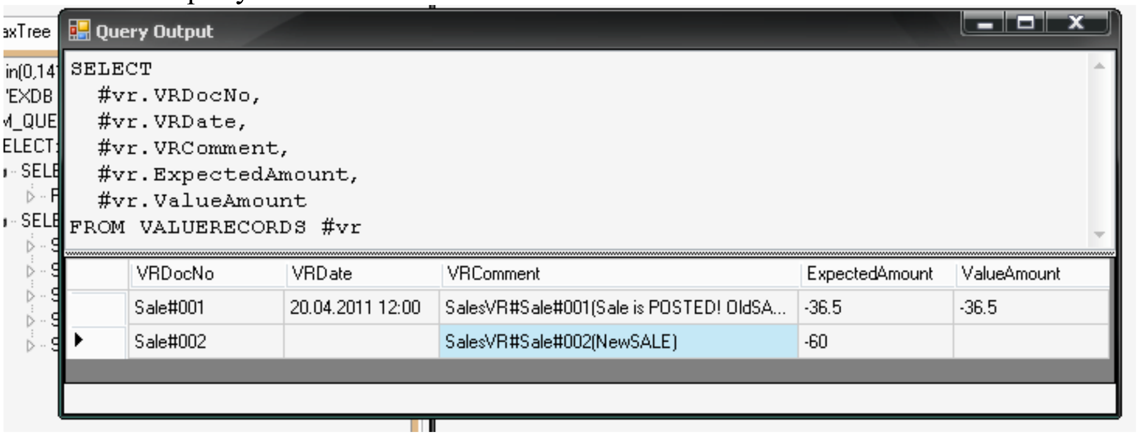
This is new result of other query



```
SELECT
  #a.Art,
  #a.Pieces2IN,  // pieces to be received
  #a.Pieces,     // pieces on stock
  #a.PiecesRes,  // pieces reserved to be shipped
  #a.PiecesFree  // pieces free to ship
FROM GOODS #a
```

Art	Pieces2IN	Pieces	PiecesRes	PiecesFree
Hat03		5		5
Trousers05		5		5
tShirt01	20	5	15	-10

Let's execute query to class `VALUERECORDS`.



```
SELECT
  #vr.VRDocNo,
  #vr.VRDate,
  #vr.VRComment,
  #vr.ExpectedAmount,
  #vr.ValueAmount
FROM VALUERECORDS #vr
```

VRDocNo	VRDate	VRComment	ExpectedAmount	ValueAmount
Sale#001	20.04.2011 12:00	SalesVR#Sale#001(Sale is POSTED! OldSA...	-36.5	-36.5
Sale#002		SalesVR#Sale#002(NewSALE)	-60	

Components of this class were realized in child class `SALES`. As a result the query returns data from objects of this class according to realization.

> Conclusion.

The RxO-prototype reaches a new level of traditional client-server architecture when the server is the client-independent environment allowing both creation and persist existence of manageable object model of an enterprise. Ad-hoc queries are used to get data describing the model state.

Offered approach has no restriction to use traditional SQL table structures. Offered language constructions are superset of existing versions SQL. It makes possible evolution of current SQL-systems, allows using an existing SQL-code and preserving existing client applications and DB access protocols. Attention can be paid to simplicity of both the approach and language expression realizing it.

The offered approach is certainly opposite to idea realized by OODBMS and ORM systems which try to realize persistence of objects existing in client application written in traditional OO language. Here client applications have to interact with other objects described by other language and existing in other system. However it exactly allows creating independent enterprise model which is accessible from many different systems. Also simplification of client applications can be supposed if they use full, active and independent enterprise model.

Data independence is one of the major priorities on all history of information systems development [4]. The RxO-system pursues the same aim interpreting the term "data" both as values and as associated functionality which reflects enterprise variability and its internal dependences. Old principle "divide and rule" can be actual still.

References:

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik "The ObjectOriented Database System Manifesto", In Proceedings of the First International Conference on Deductive and Object-Oriented Databases, pages 22340, Kyoto, Japan, December 1989

[2] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, Ph. Bernstein, D. Beech, "Third-Generation Data Base System Manifesto", Proc. IFIP WG 2.6 Conf. on Object-Oriented Databases, July 1990, ACM SIGMOD Record 19, No. 3(September 1990),

[3] H. Darwen and C. J. Date, "The Third Manifesto", ACM SIGMOD Record 24, No. 1 (March 1995),

[4] Donald D. Chamberlin and other. "A History and Evaluation of System R", Communication of ACM, Oct. 1981. http://www.cs.berkeley.edu/~brewer/cs262/SystemR.pdf