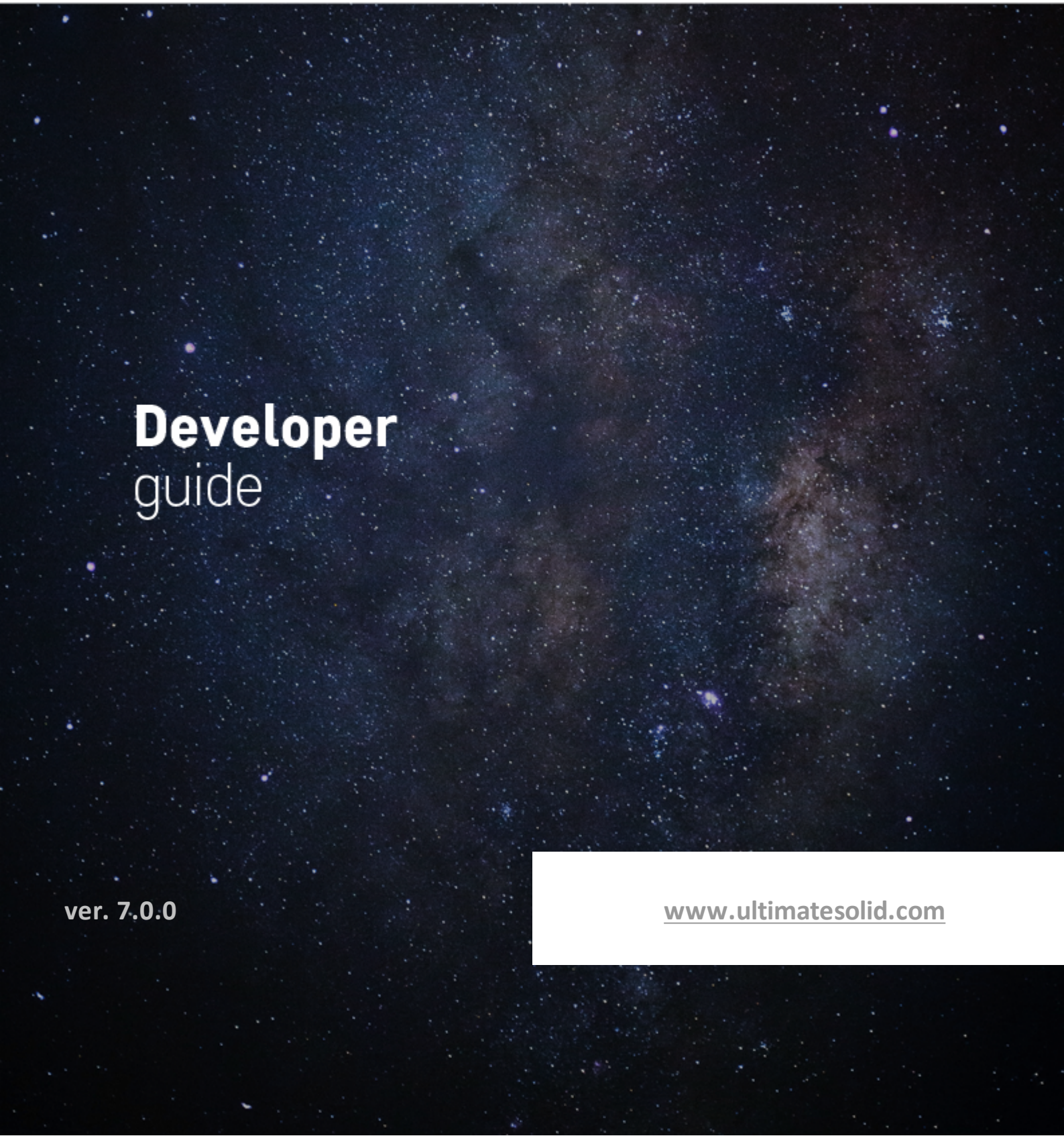
ULTIMATE SOLID

2017 R1



Developer guide

ver. 7.0.0

www.ultimatesolid.com

Table of Contents:

Introduction

On correct terminology practically applied

For the mutual mapping concept a special terminology is used.

However, since dia$par is intended for a wide range of business users, this terminology comprises well known notions like "documents", "results", "reports", etc.

Though, the familiarity with terms used in dia$par should NOT mislead.

The meaning of theses terms is much more wider in the context of dia$par.

A detailed description of the terminology used to describe mutual mapping and dia$par takes a several pages of plain definitions, that is why only some key notions regarding the managing meta-system are presented below.

Meaning in mutual mapping	Terminology of mutual mapping	Business meaning	Terrirclcgycfciia$par
The process within the enterprise cybernetic model which is determined by all possible changes pertaining to this process	mchain	The value chain notion is used in its traditional meaning	ccounentty pe
A basic feature of an individual change determining the current status of the delta of the process	mstep	A value chain stage is also used in its common meaning	cc c

			u n e n t s u b t y p e
It is a structured minimal and sufficient package of changes in measurable indicators for the cybernetic model for a specific process.	mpack	It is a small container unit of logically connected objects (that can be empty as well) which are moving along the value chain (mchain), through its stages (msteps), and transforming their nature up to becoming objects of another nature. These objects are a raw material at the input of the value chain and finished products at its output (from the local value chain point of view). An M-pack can contain any amount of information about any objects which are simulated in the dia$par.Matrix cybernetic model. Likewise, the rules of business logic applied to mpacks at every stage (mstep) of the value chain (mchain) can use values of any parameters and features of cybernetic model objects. A shipping list is a primitive example of the mpack (hence the name of dia$par has been chosen).	c o c u n e n t
The hyper-surface is a projection of the cybernetic model in a space with a less number of dimensions	mface	The Mface is a sample container containing the current values of a random number of numeric parameters of an individual section of the cybernetic model along with the history of their evolution. The number of mfaces which the cybernetic model can run simultaneously is theoretically unlimited. The population of mfaces of the cybernetic model contains all up-to-date data about the state of all enterprise parameters measured in real time.	t o t a l

It is a structured atomic package of changes for one or two mfaces representing the delta of the state which is set by the mface	equant	It is a quantum of the state of the cybernetic model. They are generated by traflexes (they are kind of conditioned reflexes of mutual mapping, please see below) An accounting transaction is a primitive example of equant.	transaction
It's random relational algebra functions projecting a subset of the model into the calculated hyper-surface.	xface	Aggregated data sets generated by low-level tools of mmizer by inquiry from developers applying data transformation techniques for a random number of basic mfaces. They are used in certain situations.	complextotal
Functions to completely transform the state of the model	mflex	Mutual mapping prescribes eight types of model responses which are similar to reflexes of a biological organism. It's interesting that all mutual mapping reflexes are, on the one hand, conditioned, i.e. they can be (and are expected to be) changed. At the same time, they are unconditioned. Because, from the point of view of the current model layout, these reflexes are 'inherent'. In other words, they were embedded by the programmer 'Creator' at the time this very model layout emerged (as a Corporate Intelligence personality).	Script
on-DEmand mFLEX	deflex	It's the model response to actions performed by a human user using visual interfaces of dia$par client applications.	connaro

SHEduled self-executing mFLEX	sheflex	The model's response to regular events	task
from the Outside calling mFLEX	oflex	The model's response to inquiries from external cybernetic systems transmitted via digital interfaces (which are mainly web services and binary protocols)	webservice
SPEcified mFLEX	speflex	A particular model's response to changes in the mpack state which depends on the mchain and mstep within which the changing mpack is located.	eventhandler
BAsic mFLEX	baflex	An unconditioned model's response to random changes	concreteventhandler
TRAnsaction mFLEX	traflex	The model's response to removing an mpack to another mstep (mchain) which describes the process of structuring and data projecting	train

		for the removing mpack container into a set of mfaces.	sactionscript
Remotely executable mFLEX	rflex	The model's responses are carried out by meta-system duplex effectors through different client applications, as well as unmanned dia$par functions which operate manufacturing equipment.	clientscript

Mfaces and mpacks in their totality are similar to computer's RAM memory and its hard drive.

The data loaded into RAM to some extent duplicates the data stored on the hard drive, but RAM memory is intended to provide the CPU with data in real time. The data stored on the hard drive is much more exhaustive, while the access time is not that critical for it.

If different access time for data stored in RAM and on a hard drive results from a structural differences of the two devices, in case of dia$par the difference between mfaces and mpacks is determined by different data packing methods (the starlike data storage model).

Mface analytic sections can be generated in any amount based on any data set from the cybernetic model, but in practice mfaces are usually based on actively used facilities of the enterprise or groups of counterparties.

An Mface has its own structure, measures and indicators of any complexity in terms of their number.

For example, an mface with data on warehouse capacity usually has two indicators — the physical capacity (in items, kilograms, liters...) and capacity in terms of money.

Thus, any equant is measured within such mface in terms of both its quantity and cost, otherwise it will just NOT be replicated by the mmizer.

Apart of spreading correct terminology, it is planned to be translated with new versions of dia$par.

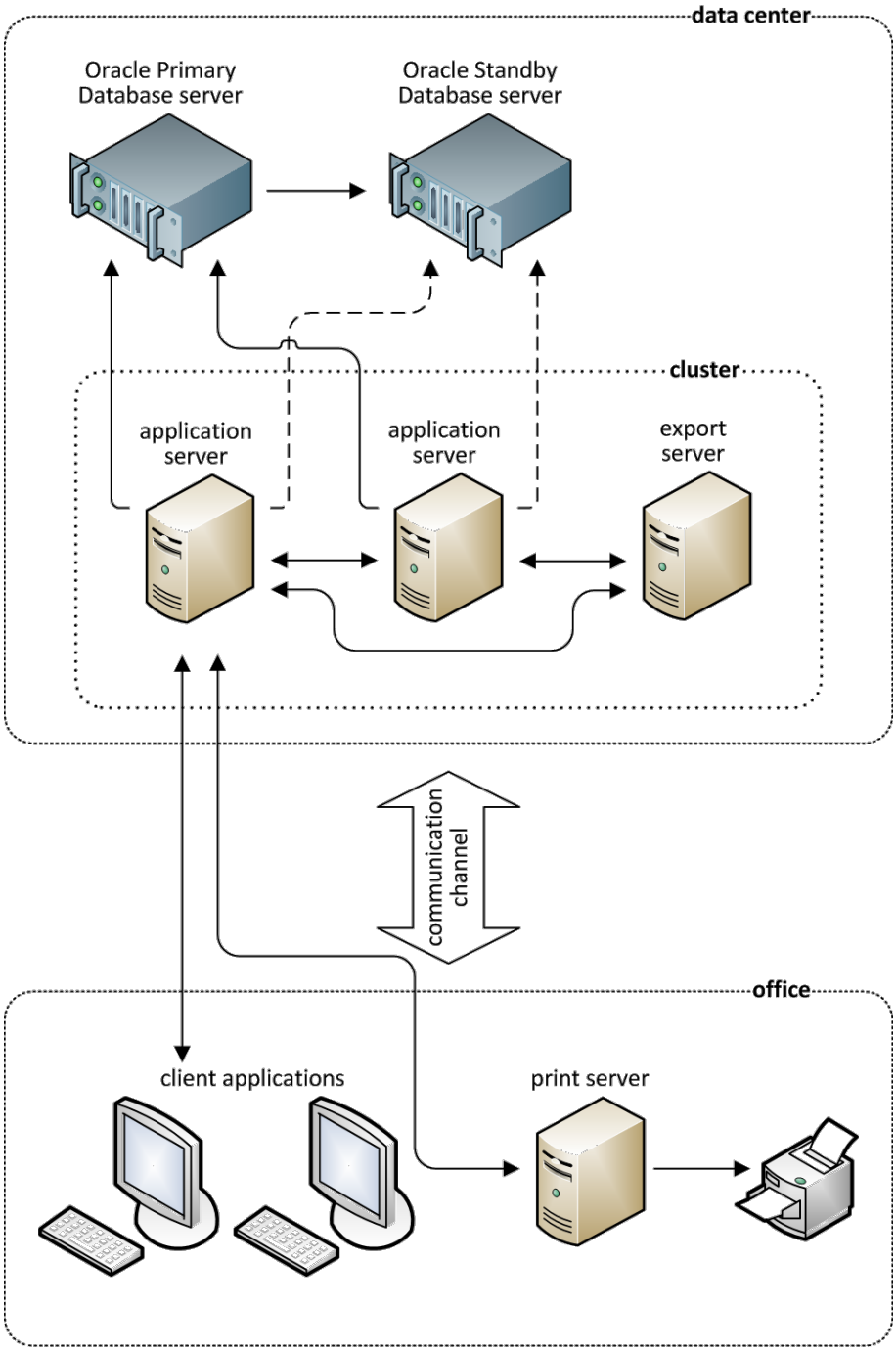
Now the mutual mapping terminology is recommended to be used by both dia$par integrators in their descriptions of target models and process engineers of companies dealing with changes in value chains.

Ultimate AEGIS® Architecture

Ultimate AEGIS® is a three–tier architecture software suite (wikipedia) which includes:

- database server (Oracle 11gR2 Enterprise Edition or Oracle 12c Enterprise Edition);
- application server;
- print server;
- client applications.

The general model of interaction is presented in the diagram below:



As a rule, and that is highly recommended to, the **database server** is located in the data center. To increase hardware fault tolerance and mitigate the risk of data loss, as well as to share the load, it is recommended to install a standby server/server cluster.

Based on similar considerations of fault tolerance and performance, the **applications server** which carries out processing of business logic and directly exchanges data flows with database server, can also be scaled into a cluster. Due to extremely high intensity of data exchange between the application server and the database server, they should be located in close proximity to each other, at least within the same local network.

As opposed to the application server and the database server, the **print server is located** at the same place with the printing devices and staff, thus reducing the load on the communications due to transferring smaller volumes of data.

With help of screen logic of **client applications,** the user can view, enter and edit data.

Composition logic of Ultimate AEGIS® - based solutions

Operated by a customer intelligent enterprise management system on Ultimate Solid platform is logically divided into two parts:

The Platform Ultimate AEGIS® **is an unambiguous part** (hereinafter the terms “platform” and “kernel” are interchangeable), which can only be changed by the vendor (the Ultimate developers). In particular, it ensures operation of the second part called “business logic space” which is ambiguous and can be changed by either partners or independent developers.

The business logic space contains scenarios of business rules processing, business rules themselves, screen forms and so on and so forth. The specific contents of the business logic space corresponding to a particular client or (which is a bit more general) to a particular business sphere is called “ business logic space configuration” or just “configuration”.

The Installation is a fully deployed software which is the platform and an adapted configuration for a specific client. A ready-to-use set of the platform and configuration is called a solution which is usually named after the corresponding configuration. For example, for e-Trade configuration the solution will be also called Ultimate e-Trade. Henceforth ,we shall use the terms “system”, “product” and “solution” as synonyms.

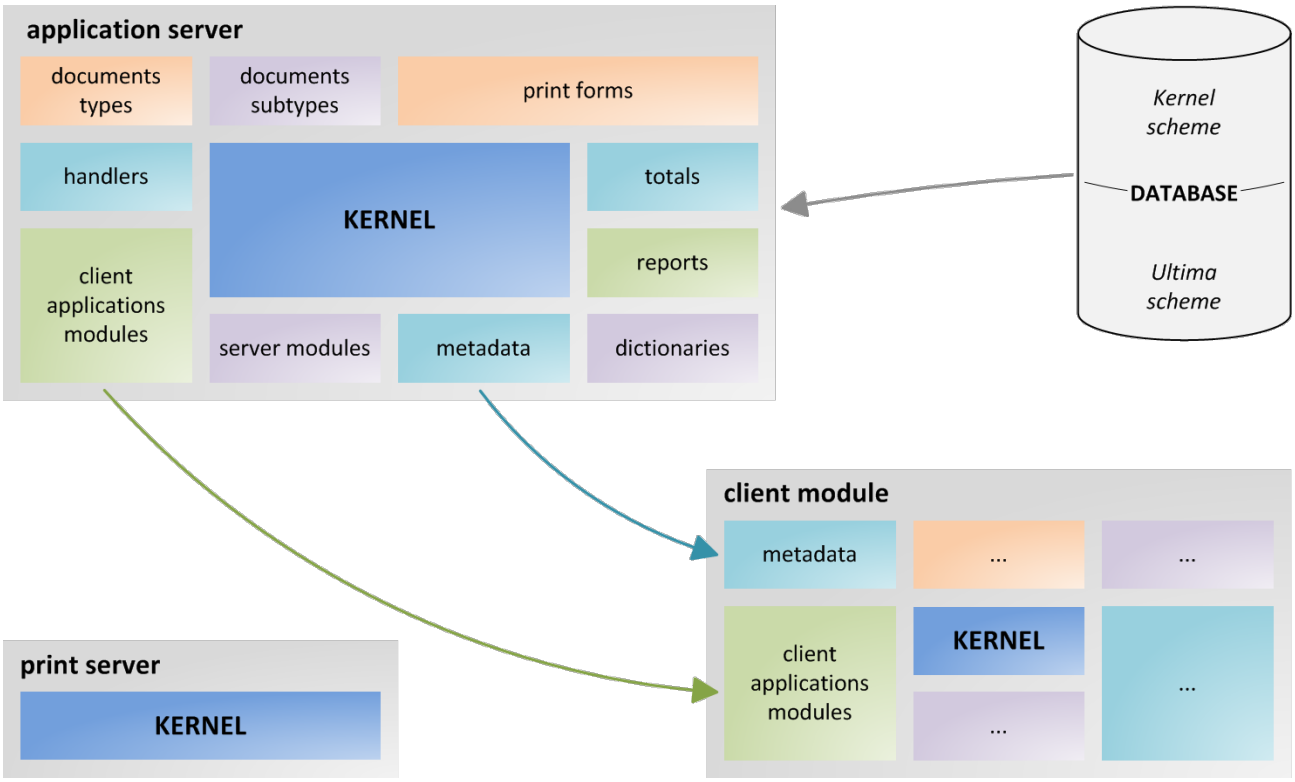
The platform ensures the functioning of the entire software system as a whole and provides the following tools and mechanisms:

- Users management , their authorization and entitlement inspection;
- communication between applications;
- access to the DBMS (database management system);
- configuration changes (version control);
- conversion of “raw data” from DB to system objects;
- integration services – SOAP, JSON, REST, XML, etc.;
- development environment and business logic changes.

Configuration is a set of metadata describing the structure of business objects and a specific data:

- documents;
- dictionaries;
- links and references of documents and references;
- dictionaries;
- handlers, describing the logic of interaction between business objects;
- server modules;
- client modules;
- user rights.

The general model is presented on the chart:



The configuration is stored entirely in the database. The supported databases include Oracle 12c Enterprise Edition and PostgreSQL 9.6. The database contain two schemes:

- *Kernel* – its structure is static. It is the very scheme the application server loads configuration from when launched;
- *Ultima* – modifiable by any developer, in which tables are created and data of business logic space is stored.

PostgreSQL database uses a few more schemes, also owned by the kernel. PostgreSQL-based version has a few minor limitations compared to the Oracle-based one. Except for these limitations, the two versions are functionally equivalent. For more information about the limitations and the development process of the PostgreSQL-based version, see the dedicated chapters.

The kernel schema structure is unavailable to the changes by application developer, instead it provides an interface and tools to implement business logic. The business logic performed on the application server is implemented in the form of classes in C# inherited from the corresponding classes of a platform (further such classes will be called scripts). The screen logic is carried out on the client application and is implemented in the form of modules on any .NET compatible language.

The system provides to the applied developer ready to be used:

- authorization and authentication mechanisms;
- mechanisms of conversion of objects, their unloading and saving in to the database;
- editor of business objects and business logic;
- communication mechanisms between parts of the application;
- mechanisms for rapid prototyping of user interfaces;
- system updates without restarting, and more.

Task of the application-oriented developer is creation by means of these tools new and/or modification of existing configuration which represents set of reference manuals, documents, outcomes, processors, screen forms, etc., and also their descriptions.

Developer

First steps

Means of the subject area description

The system suggests formulating the description of the subject area by listing the dictionaries, link tables, documents, totals and scripts that handle various events in the system.

Dictionaries and **link tables are meant for storing the static and rarely changing data**. It will be ok to present them as simple tables.

This is one of the simplest objects in the system. For example, the list of products sold or manufactured by the company can be a good example if presented as a dictionary.

As soon as the system recognizes the description of a new dictionary (its name, a set of fields, etc.), the developer can immediately use a form to create the list of the dictionary records and a form to edit the records. This allows quick move to the implementation of other logic, without losing time to develop the interface. The link tables are meant to the many-to-many link. The description of the link table properties allows the system automatically generate the ready data management interface and offer it to the user or the developer.

Documents and **totals** are the objects that are more complex.

Documents contain the information about a certain event in the subject area (for example, the sale of products).

Documents are the composite objects, unlike dictionaries. Each document contains a standard header (the set of fields that is present in any document of the system), an individual header — the set of fields that exists only in this document, and a set of table parts (it can be empty or not — the number of the table parts for one document is not restricted, and this simplifies the formulation of the complex business logic to a great extent).

A table part allows keeping a list of records within a single document (rows listing products enjoy the highest demand here, but it isn't the only example). The documents are closely related to the concept of a total.

Total is the object of the system, which allows describing the measured parameter of the subject area and keeps the record of its changes. The total can be also presented in the form of a multivariate cube: the same terminology – dimensions and variables is used in the system. Thus, the document retains the information about the event and converts its data into a set of the totals' changes. An example of a total can be the in-stock products at the storehouses. This system object stores the information on the quantity of each product remaining at each storehouse. Accordingly, the sale document reduces the entire total of the products remaining at the storehouses (for the products in the document).

The following terminology will be used further on:

- *dictionary* (a synonym for a *dictionary type*) — the dictionary type defines a set of fields and other properties;
- *dictionary record* — a specific record (or a specific object of the system) in a dictionary;
- *link table* (a synonym for an *link table type*) — similar to a dictionary, it defines what fields are present in a link table;
- *document type* — the description of the set of fields in a document header and other properties;
- *document* — one particular document containing information about an event.

Dictionaries

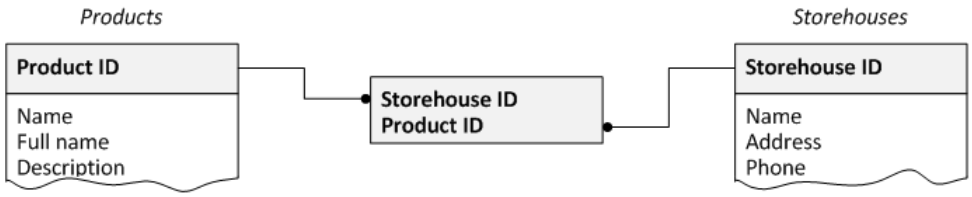
Dictionaries store the information about the objects of the described model.

The system generates (for each dictionary) a class to represent a dictionary record. In addition, the system generates the SQL script to create the necessary tables (and other objects for the dictionary storage) in the database. Accordingly, each dictionary contains the records which fields can take the values of the following types:

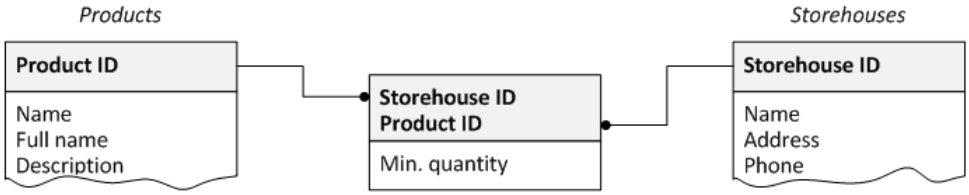
- *numbers* – even and fractional (different types use different visual display);
- *strings* – short (2048 characters) and long (the size of which is limited only by server disk space);
- *dates*;
- *logical (Boolean) type*;
- *binary data* (e. g. photos);
- *references* to other dictionaries or documents.

Dictionaries can be of flat and tree types. They differ in their form of displaying the content – in a table or tree form, respectively.

Link tables are used for keeping the links between the dictionaries in the system. Link tables do not have their own form of display, but their data is available for viewing and editing in the screen forms of those dictionaries, which they link. For example, they help to set the storehouses/products relations, when a large range of products is stored at many storehouses:



Apart from keeping the links between the dictionaries, the link tables can also be a storage of some additional values. An example of this can be the situation when, in addition to the storehouses/products relations, it is necessary to store the value of the minimum required reserve stock amount at the storehouse:



Totals

Totals contain information about the current state of the company's measured performance indices, as well as the history of their changes. Each change of a total is called a **transaction**. If drawing an analogy to the accounting practices, it can be said that totals are the distant descendant of the book accounts concept, while a transaction, respectively — of the accounting records.

Totals consist of dimensions and variables. Each dimension of a total is a reference to a dictionary. Totals can have many dimensions, thus being multidimensional cubes.

Transactions show what changes occur with respect to each variable, and on what dimensions.

The totals can be **balance and non-balance**.

The double-entry rule applies when introducing changes to a balance total: then every transaction is the transaction in pairs (with the opposite values-debit and credit) and their sum is always zero.

Double-entry is not applicable when introducing changes to the outcome of a non-balance total. Accordingly, one can not make a transaction between the balance and non-balance totals.

The system can automatically generate a standard form of **report**, for each total; this form has customizable grouping and filter mechanisms.

It might seem (at first glance) that the totals do not differ from the link tables mentioned above.

However, it is not so.

The difference is that the values stored in the link tables, are not the measured parameters; they are set manually. If you want to implement the totals by using the link tables, you will have to develop the mechanisms for their completion (as well as the reporting forms and other tools) right from scratch.

Documents

Documents contain information on current events, which are the basis for generation of **transactions** – changes in totals. Similarly to the dictionaries, the system generates a class for each type of the document to store a single document.

Type of documents specifies:
- set of fields in the document header;
- set of table parts;
- set of document subtypes;

A document consists of:
- a common header – a set of fields typical for all document types;
- a header – a set of fields typical only for the given document type;
- a set of table parts – a document's massive of data of the same types that characterize dictionaries.

The **table parts** themselves are separate objects and shall be described separately.

The document type determines the set of table parts types it includes. A document may include two table parts of the same type. Similarly to the dictionaries, the system generates a class for each type of the table part to present a single record. Unlike the dictionary, a record of a table part has a predefined set of fields, which allows the record to be linked to the document.

Need for any document type is determined by the business logic. This can be, e. g., receipt documents, cash documents, inventories, etc.

An event, information on which is stored in the document, can be stretched in time, for example, sale occurs through the stages of order, picking up at a store, payment and release. If the information on intermediate stages is not so important, the stages can be presented as **document subtypes** for the benefit of users or developers at the sacrifice of accessibility of such information. A state is similar to a document subtype. A developer designates possible subtypes for each type of documents. In addition to the description of domain knowledge and representation of the current state, a system administrator distributes rights to separate system subtypes, e. g., he can grant rights to execute all operations within a single subtype and "read only" rights within the rest of subtypes.

As it was said above, documents store information about events, while totals allow distributing such information to different slices. Transformation of data in a document into transactions of totals is carried

out by means of special programs – transaction scripts, or posting handlers, coded by application programmers.

Such scripts are launched every time the document is saved, and the system will see to it that the set of transactions is saved in the database in the best way. Besides the transaction scripts, several other scripts are triggered during saving of transactions; these scripts allow checking arbitrary business rules. The triggering order described in details in the following chapters.

Consider an example of interrelation of documents, totals and posting handler's operation.

First, let's buy *article* "1" from *supplier* "1" and place it in *store* "1".

The receipt takes form by means of a document of the *purchase* type. In the document's header, a *supplier* and a *store* for the incoming articles are specified; the *articles* are to be listed in the table part:

Purchase #1

Supplier 1
Storehouse 1

Product	Quantity	Price	Amount
1	15	10	150

This process enables three dictionaries:

Storehouses

ID	Address	Area
1	82 Pine St	214

Droducts

ID	Name	Amount
1	DVD-R	20

Agents

ID	Name	Address
1	Future Simple LTD	78 Old Mill Rd

When saving the document "1", the posting handler is called and generates the transaction "504". On the basis of this transaction, the kernel makes the following changes in totals:

- in the total "*stock (transactions)*", one row is subjoined, which increases the number of *article* "1" on *store* "1" by *quantity* and *amount* of the receipt;
- in the total "*agent debt*", one row is subjoined, which decreases the debt of *supplier* "1" by the *amount* of the receipt.

before purchase

the remaining stock at the storehouse (summary)

Product	Storehouse	Quantity	Amount
1	1	0	0

the remaining stock at the storehouse (movement)

Document	Product	Storehouse	Quantity	Amount
504 → 1	1	1	15	150

the contractors' debt

Document	Agent	Amount
-504 → 1	1	-150

after purchase

the remaining stock at the storehouse (summary)

Product	Storehouse	Quantity	Amount
1	1	15	150

For the total "*stock (transactions)*", in the example above, the fields *document*, *article* and *store* are dimensions and refer to the corresponding tables; the fields *quantity* and *amount* are variables.

Now, let's sell the *article*.

The sale takes form by means of a document of the *sale* type. The document's header specifies a *client* and a *store*, where the articles for sale are stored; the *articles* are to be listed in the table part: Payment accompanying the sale is processed by a document of the *payment to cash* type. In the document's header, a *client* and a *cash* accepting the payment is specified; amount of payment shall be specified in the table part:

Sale #3

Client 2
Storehouse 1

Products	Quantity	Price	Amount
1	7	20	140

Cash-desk payment #2

Client 2
Cash-desk 1

Amount	Description
140	

At this stage, the process involves one more dictionary:

Agents

ID	Name	Address
1	Future Simple LTD	78 Old Mill Rd
2	Present Perfect LTD	11 Loring Ave

Cash-desks

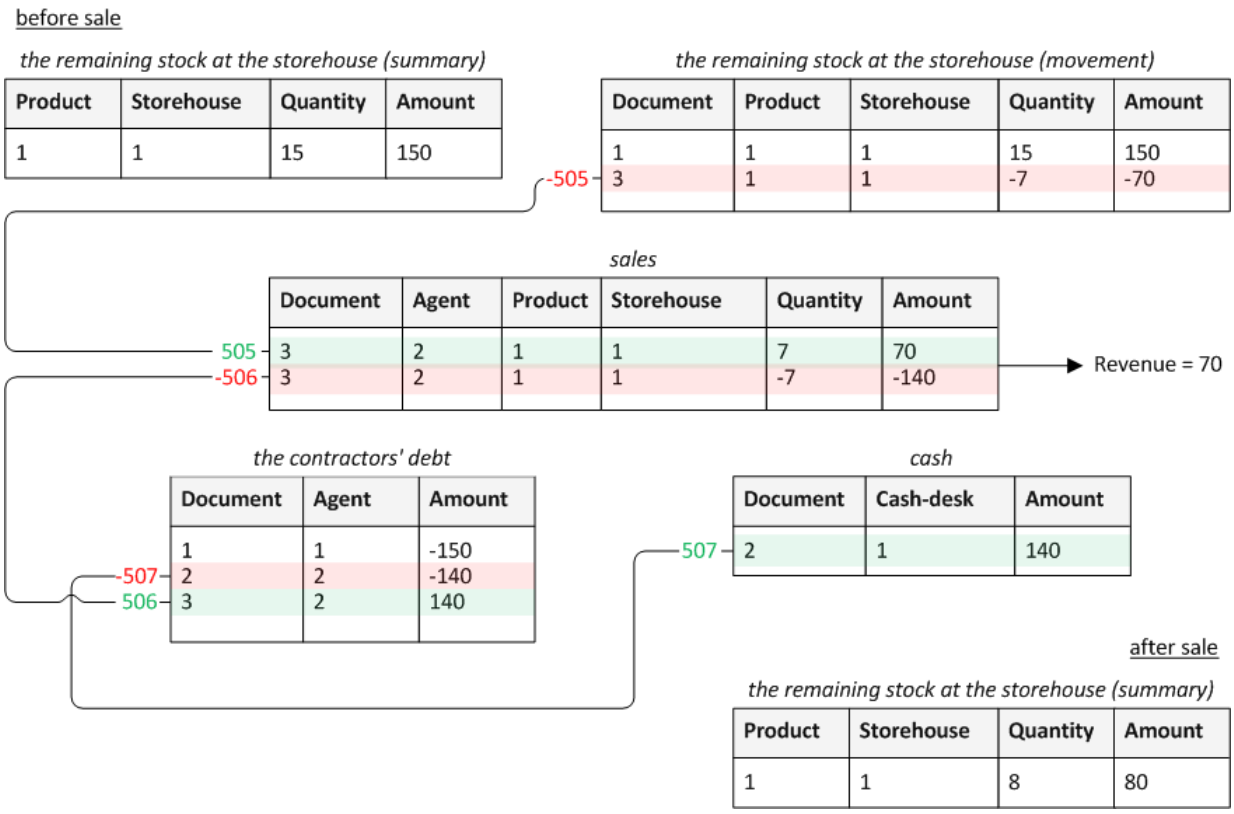
ID	Address
1	82 Pine St

When saving the document "3", the posting handler is called and generates the transactions "505" and "506". On the basis of these transactions, the kernel makes the following two pairs of changes in totals:

- in the total "*stock (transactions)*", one row is subjoined, which decreases the number of *article* "1" on *store* "1" by *quantity of* expenditure and *amount* corresponding with the cost of previous receipt;
- in the total "*sales*", a row is subjoined, which increases *quantity* of the *article* "1" being sold to *client* "2" at *amount* of sale cost;
- in the total "*sales*", a row is subjoined, which decreases *quantity* of *article* "1" being sold to *client* "2" at *amount* of sale (at selling price), thus, we have proceeds arising from the pricing spread;
- in the total "*agents debt*", a row is subjoined, which increases the debt of *client* "2" by *amount* of the sale.

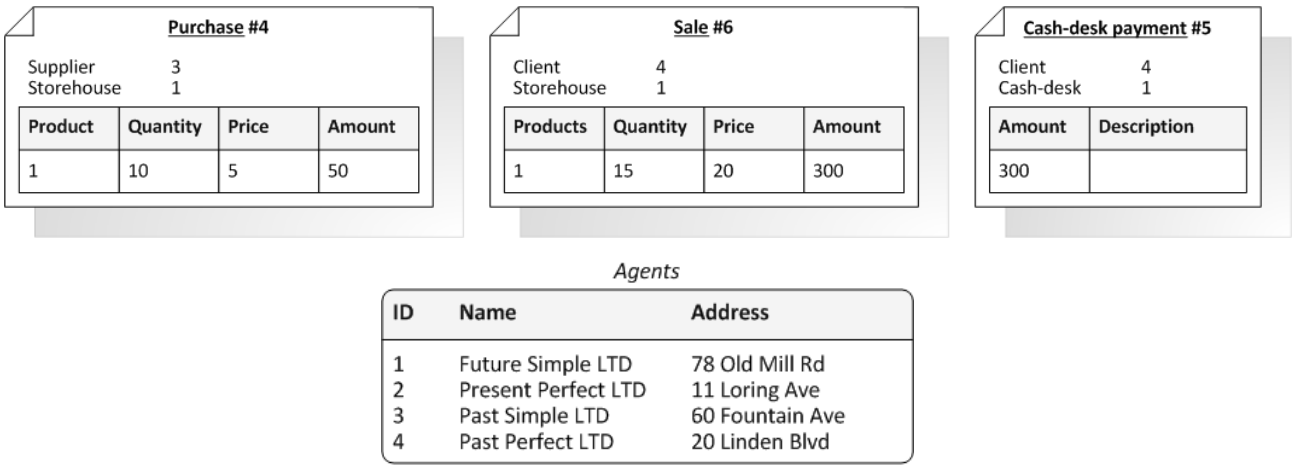
When saving the document "2", the posting handler is called and generates the transaction "507". On the basis of this transaction, the kernel makes the following changes in totals:

- in the total "*agents debt*", a row is subjoined, which decreases the debt of *client* "2" by *amount* of payment.
- in the total "*cash*", a row is subjoined, increasing the money in *cash* "1" by *amount* of payment.



In fact, due to the difference in purchase prices, the cost would be calculated not so definitely, as shown in the example.

Consider one such case. Let's buy *article* "1" from a different *supplier* "3" at price different from the previous one (document "4"), and then resell it to *client* "4" (documents "5" and "6"):



When saving the document "4", the posting handler is called and generates the transaction "508". On the basis of this transaction, the kernel makes the following changes in totals:

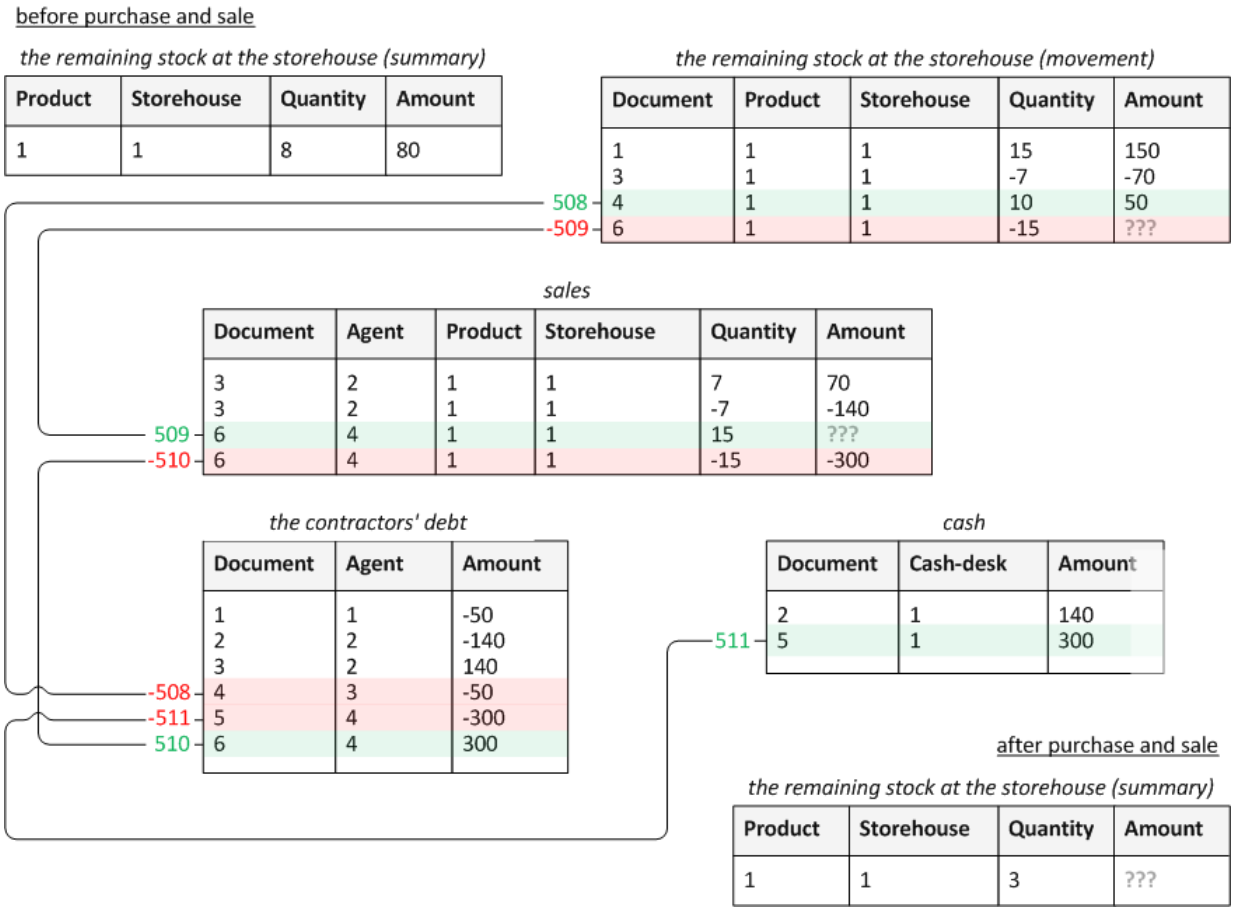
- in the total "*stock (transactions)*", one row is subjoined, which increases the number of *article* "1" on *store* "1" by *quantity* and *amount* of the receipt;
- in the total "*agent debt*", one row is subjoined, which decreases the debt of *supplier* "3" by the *amount* of the receipt.

When saving the document "6", the posting handler is called and generates the transactions "509" and "510". On the basis of these transactions, the kernel makes the following two pairs of changes in totals:

- in the total "*stock (transactions)*", one row is subjoined, which decreases the quantity of *article* "1" at *store* "1" by *quantity* of expenditure; *amount* of this expenditure, being a cost, is subject to calculation, therefore the newly-created transaction has a blank field;
- in the total "*sales*", one row is subjoined, which increases *quantity*of *article* "1" being sold to *client* "4"; the *amount* field, again, being a cost, remains blank;
- in the total "*sales*", one row is subjoined, which decreases *quantity* of *article* "1" being sold to *client* "4" at *amount* of sale (at selling price);
- in the total "*agents debt*", a row is subjoined, which increases the debt of *client* "4" by *amount* of the sale.

When saving the document "5", the posting handler is called and generates the transaction "511". On the basis of this transaction, the kernel makes the following changes in totals:

- in the total "*agents debt*", a row is subjoined, which decreases the debt of *client* "4" by *amount* of payment.
- in the total "*cash*", a row is subjoined, increasing the money in *cash* "1" by *amount* of payment.



Variables remaining blank in the process of storing the transactions are **analytical**. In case of the total "*stock (transactions)*", this is the variable *amount*. When buying, *amount* is known, therefore during posting it goes to the total. But when it comes to the outgo, the amount needs to be calculated, and the field remains blank. Whereas the variable *quantity* is **operational**; we always definitely know which *quantity of articles* is being recorded as received, written off, transferred, or sold.

To explain how the cost is calculated, we need to consider in details the structure for totals storage.

Each total is implemented with the help of four tables, two of which are **operational** and the others **analytical**. Similarly to variables, the information is put into operational tables right after the transaction

has been recorded, while into the analytical tables the information goes after the totals have been calculated. At the same time, the operational tables may contain analytical variables, such as "*stock (transactions)*". It is analytical variables of the operational total that remain blank, if the transaction doesn't include their calculated values.

The tables can be distinguished by prefixes:

- **TB**_tablename – **operational summary table**, which contains only one set of variables' summary values for each set of dimensions. In the example above, this is the table "*stock (summary)*", which contains a current balance and its value for each article stored;
- **TR**_tablename – **detailed operational table**, which includes all transactions. Having summed their variables for a certain set of dimensions, we can obtain a summary value that is stored in TB_tablename. In addition, this table contains information on documents and transactions that led to changes. In the example above, this is the table "*stock (transactions)*";
- **TD**_tablename – **detailed analytical table**, which includes the same transactions (only already calculated) as the TR_tablename. The table is filled-in after the totals have been calculated. In the example above, this is the table "*stock (analytical transactions)*";
- **TT**_tablename – **detailed analytical table**,which contains only one set of variables' summary values from the table TD_tablename for each set of dimensions. In the example above, this is the table "*stock (summary, analytical)*";

For the latter example, tables of the total "*stock*", after the cost has been calculated, will be as follows:

before purchase and sale

the remaining stock at the storehouse (summary) TB

Product	Storehouse	Quantity	Amount
1	1	8	80

the remaining stock at the storehouse (movement) TR

Document	Product	Storehouse	Quantity	Amount
1	1	1	15	150
3	1	1	-7	-70
4	1	1	10	50
-509 6	1	1	-15	???

sales

Document	Agent	Product	Storehouse	Quantity	Amount
3	2	1	1	7	70
3	2	1	1	-7	-140
509 6	4	1	1	15	???
6	4	1	1	-15	-300

the remaining stock at the storehouse (analytical movements) TD

Movement	Document	Agent	Product	Storehouse	Quantity	Amount	Incoming document
504	1	1	1	1	15	150	1
505	3	2	1	1	-7	-70	1
508	4	3	1	1	10	50	4
509	6	4	1	1	-8	-80	1
509	6	4	1	1	-7	-35	4

the remaining stock at the storehouse (analytical summary) TT

Document	Product	Storehouse	Quantity	Amount
6	1	1	3	15

after purchase and sale

the remaining stock at the storehouse (summary) TB

Product	Storehouse	Quantity	Amount
1	1	3	???

Calculation of cost shall be carried out by the handler – a **total driver**. The calculation of cost is based on FIFO principle, according to which the total driver calculates the selling cost and makes two records in the table "*stock (analytical transactions)*". An application programmer is able to code his own total driver and implement the calculation in accordance with another method, e. g., LIFO or average costs,

though such methods would be a mere tribute to the prehistoric times of the management accounts, when an accurate calculation by FIFO was impossible due to inconceivable efforts.

As a rule, totals are calculated automatically every hour, but the relevance of totals may be even less exact: if corrections are introduced into a document causing the transactions to change the totals post factum, the data will be relevant up to the date of this document.

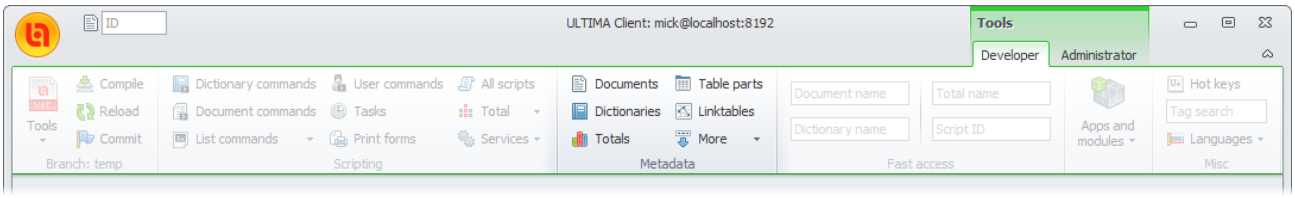
In this connection, a report on totals may be relevant by the date less than the actual date. Information on the date of the report relevance is displayed in its form view.

How to make simple metadata objects

We will consider creation of simple objects of metadata on examples.

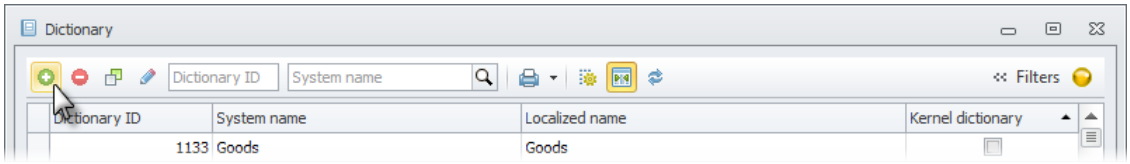
Starting with this chapter the installed system Ultimate AEGIS® will be required. Users with the right Developer have to be created by the administrator for developers. In formation about the installation and control in detail it is possible to find in the *guide of the administrator*.

After starting the main client application (ClientLoader.exe) the user, who has the right Developer, will see the additional tabs in the main menu. Metadata are described by means of the dictionaries located in *Metadata* group in the "*Developer*" tab:

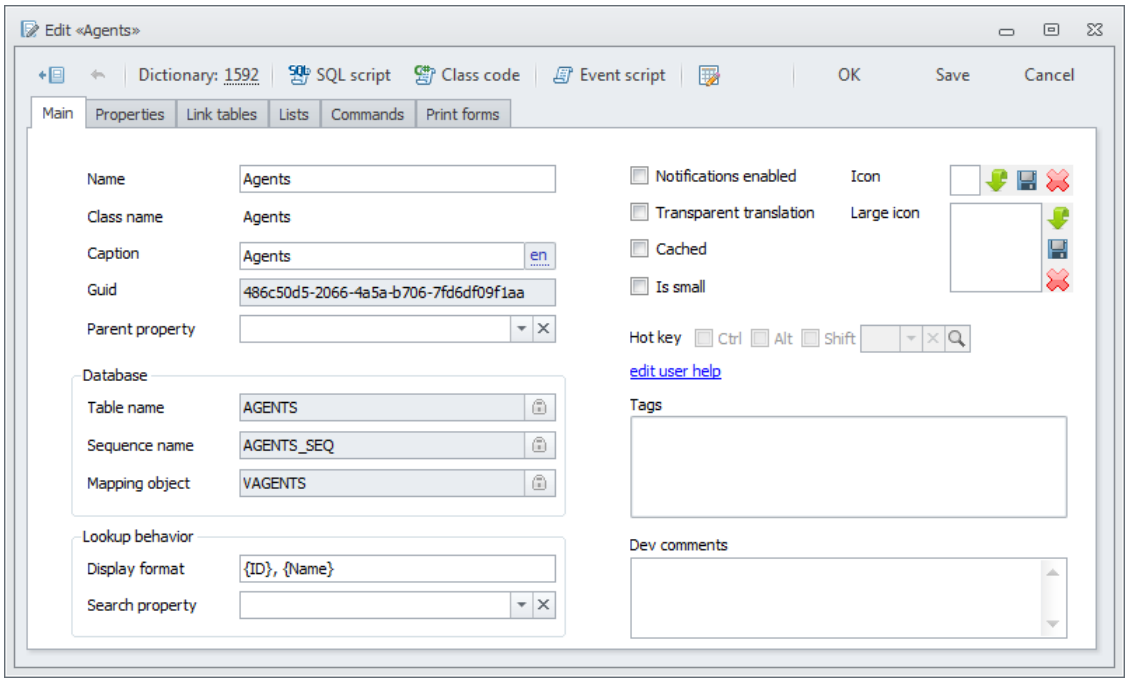


How to make dictionary

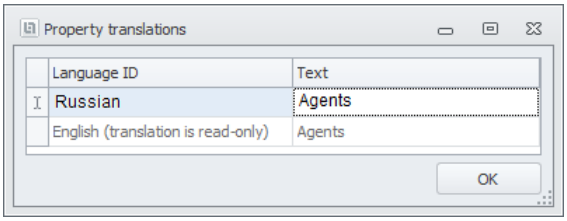
In the list forms of dictionaries (menu item Dictionaries), create a new record:



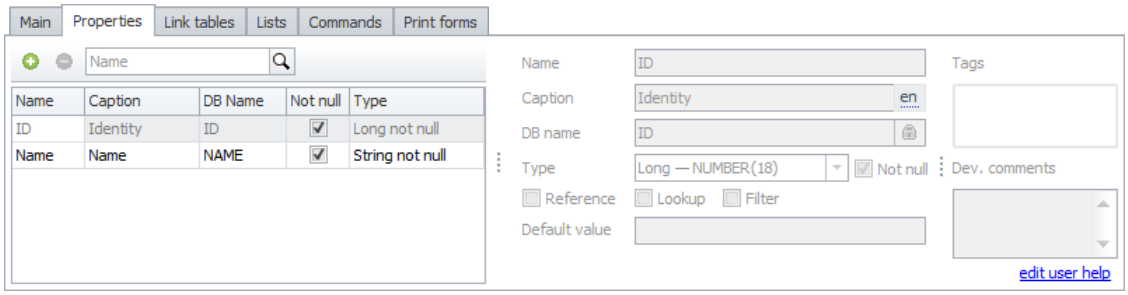
In the *Main* tab, set main parameters. *Name* of the dictionary defines the name of its objects in the database. Format of dictionary record *Display format* is responsible for how dictionary records will be displayed in a short type, for example, in management elements :



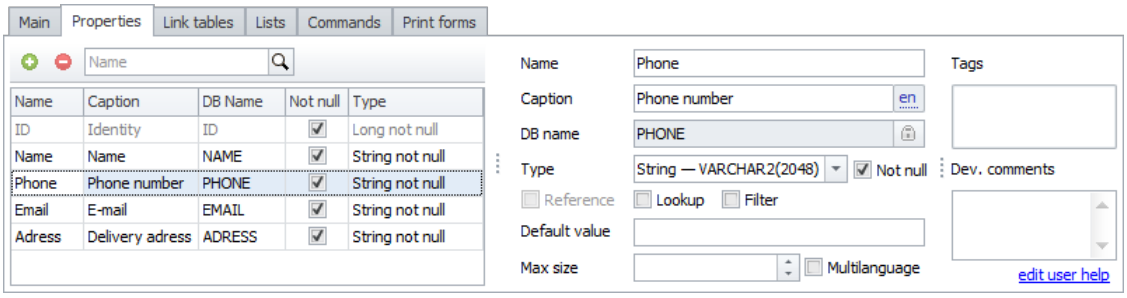
Caption description will be displayed in the screen forms as the dictionary name. This multilingual property (has its value for each of the languages supported by the system):



On *Properties* tab, set the properties of dictionary records. Two properties – identifier of the dictionary record *ID* and its *Name* – are created automatically, and the identifier cannot be removed or changed:

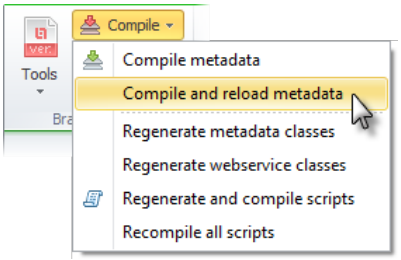


For each created property it is necessary to specify *Name* that determines names of its object in the database (value field *DB Name* will be generated automatically), description *Caption* and data *Type*:

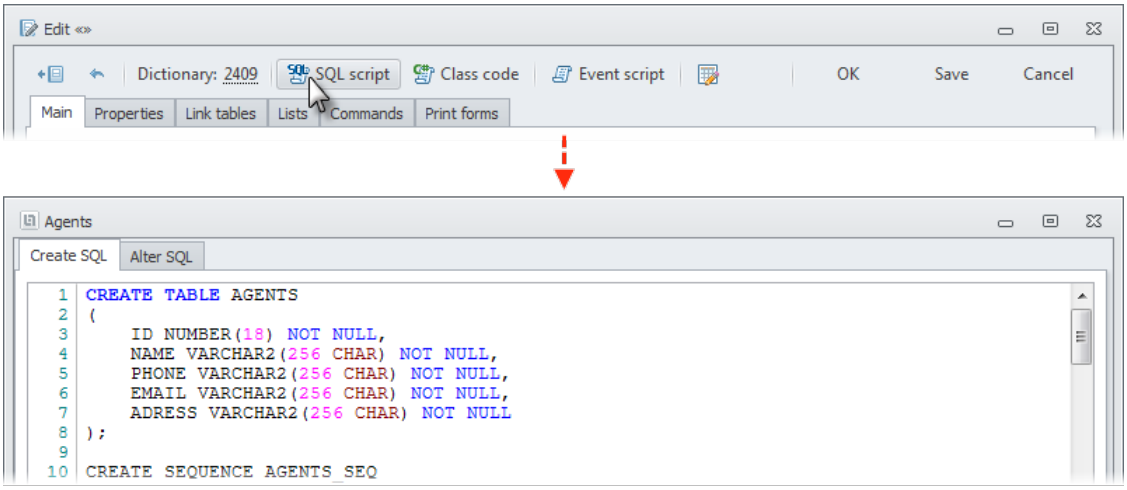


Now it is possible to save the dictionary.

When saving the new dictionary its class will be generated in the system Ultimate AEGIS®. After compilation and reset of metadata the generated class can be used when writing, for example, scripts or handlers. Also the command of opening of a dictionary list-oriented form will be available to add in the main menu.



The final step will consist in creation of objects dictionary in DBMS in *Ultima* scheme by means of the SQL script generated by application Ultimate AEGIS®:



However, at first for the dictionary it is necessary to give the rights for it to the user for successful completion of this operation.

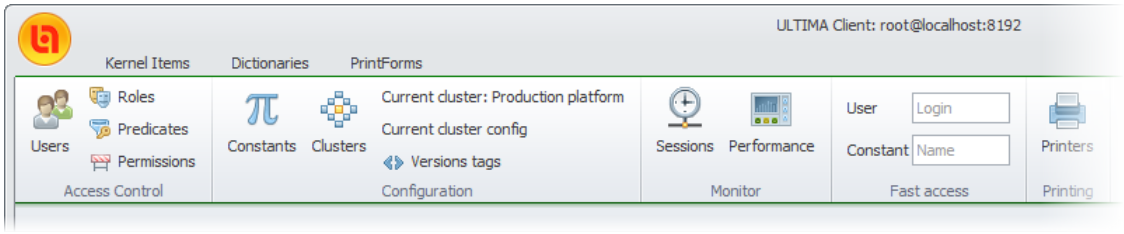
After all the script needs to be executed in the *Ultima scheme* in any application for work with the relational databases supporting SQL, for example, PL SQL Developer and TOAD. Just before the script execution it is necessary to specify on behalf of what user and on what branch of meta data changes are made For this purpose it is necessary to execute KERNEL.SET_LOGIN method having specified the user login to whom the dictionary rights and an application server code were given:

```
EXEC KERNEL.SET_LOGIN('UserLogin', 1)
```

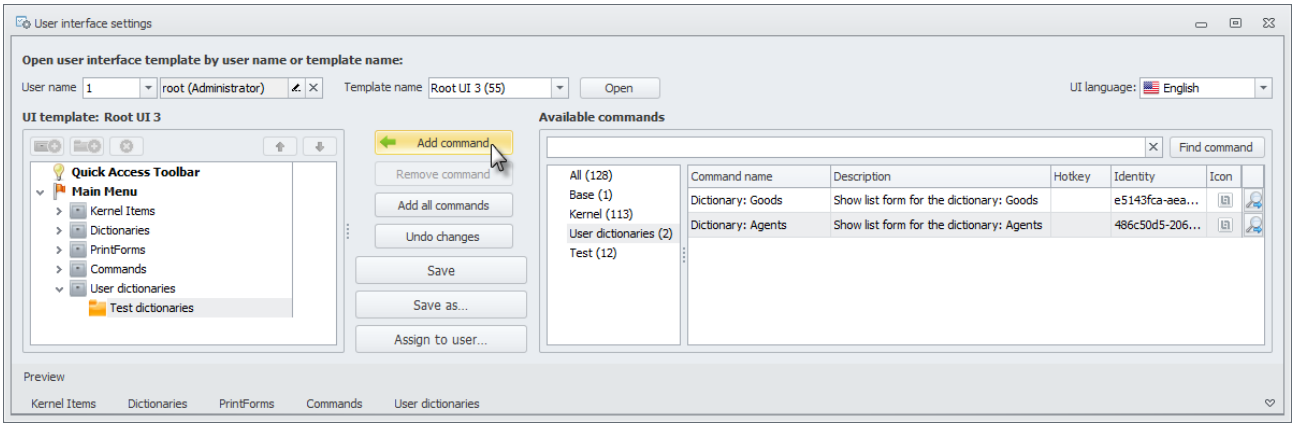
Adding a dictionary to the interface

In order the user could open a dictionary list form, an option should be provided to fetch it.

In addition to combination of hot keys, the dictionaries can be opened via main menu:



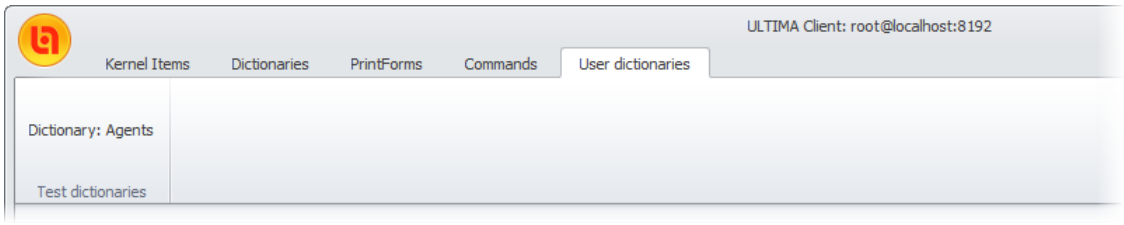
To add the created dictionary to the main menu, use a form for setting of user interface (item *UI settings* of menu):



The interface template should be edited for the user, who requires provision with a possibility to call a command (in our case, it is *Administrator* with login *root*).

All user dictionaries including newly created one can be found among the objects of the group *User dictionaries*.

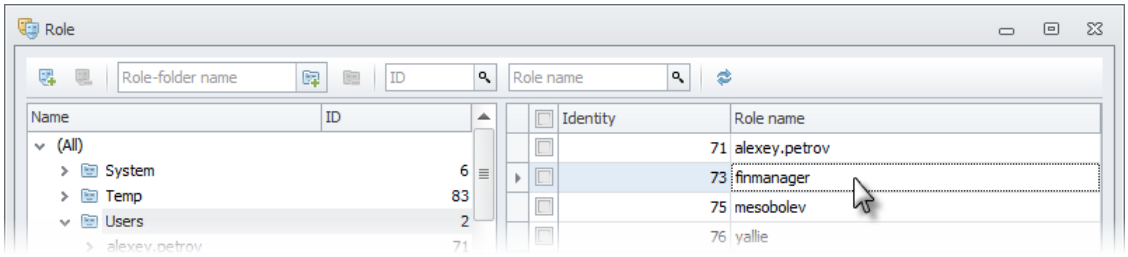
After adding a dictionary to interface template and its saving, it becomes available in the main menu:



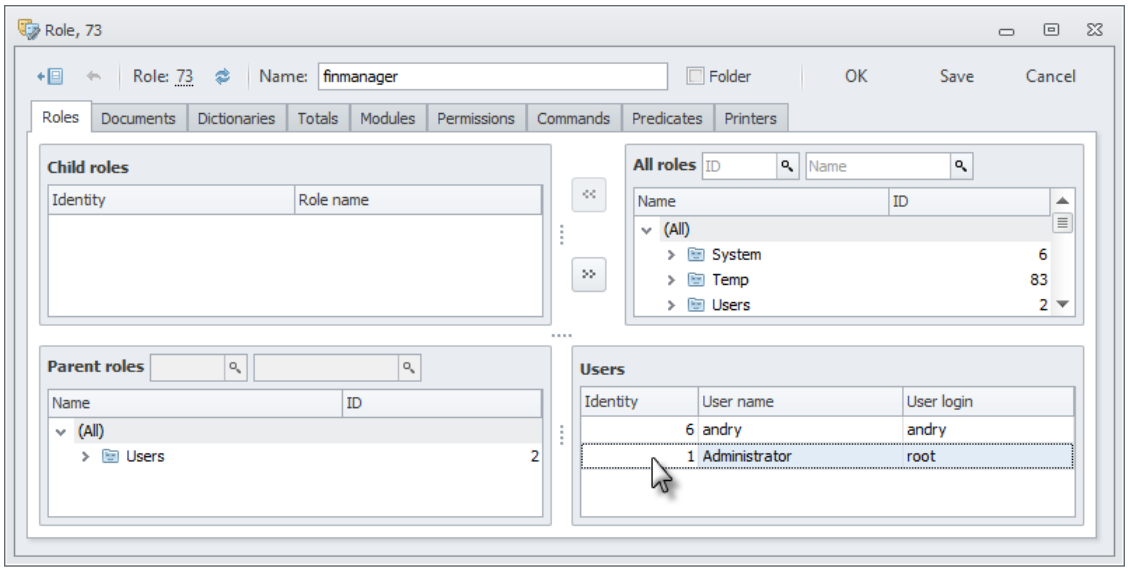
Issuing of permissions

In addition to the option for fetching of dictionary list form, a user must have corresponding permissions for its opening.

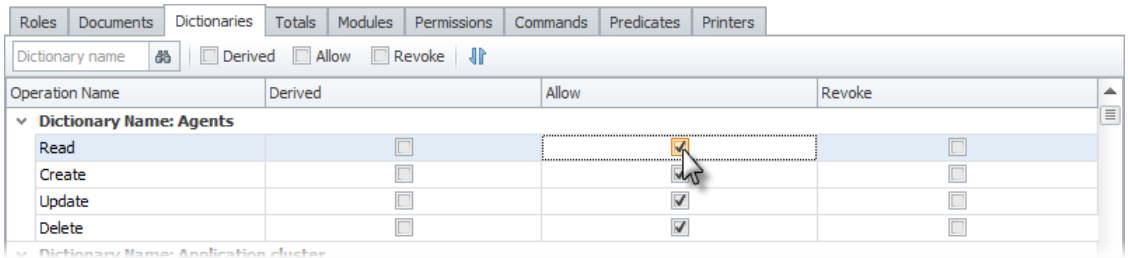
For that purpose, in the roles dictionary find a *Role* of the user, who should be provided with a possibility to access the dictionary:



In the Roles tab, you can make sure that the edited role is assigned to the very user, whom we are going to grant access to the dictionary (in our case, it is *Administrator* with login *root*). Right there you can see, what other users are, who can be assigned with the role, and who will be also granted that access with, correspondingly:



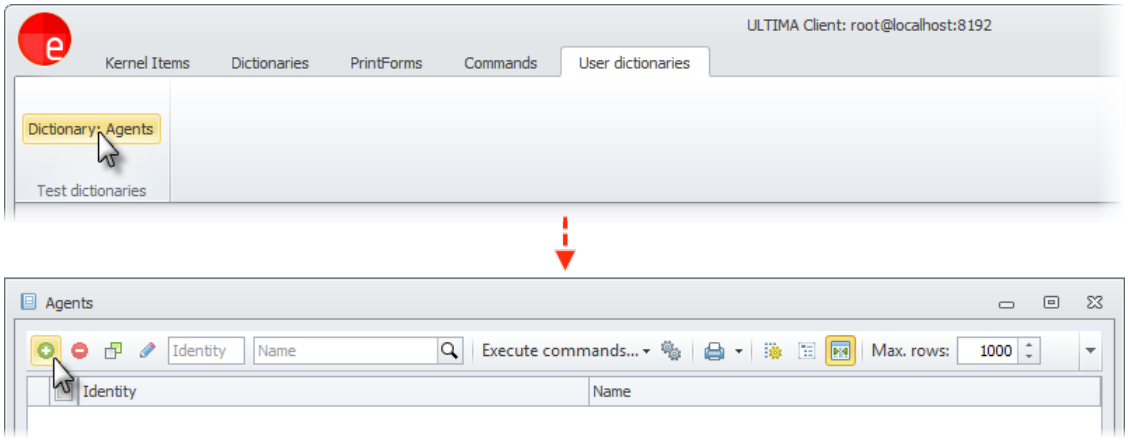
In the Dictionaries tab, grant access to the dictionary. In our case, it is full access, while *Read* access is sufficient to open the search form and view dictionary records:



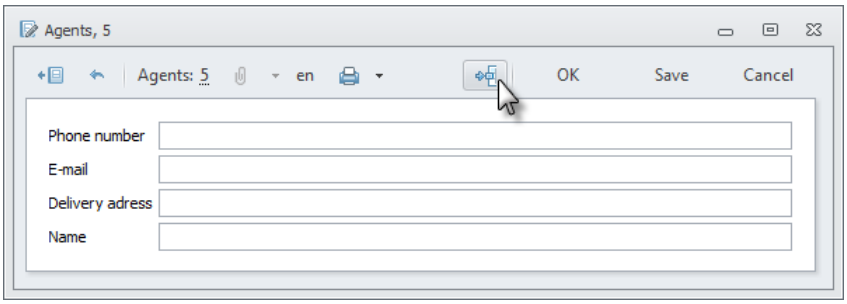
Now, after saving the modifications made to the role, *Administrator* has corresponding permissions to open the dictionary list form.

Editing of standard dictionary screen forms

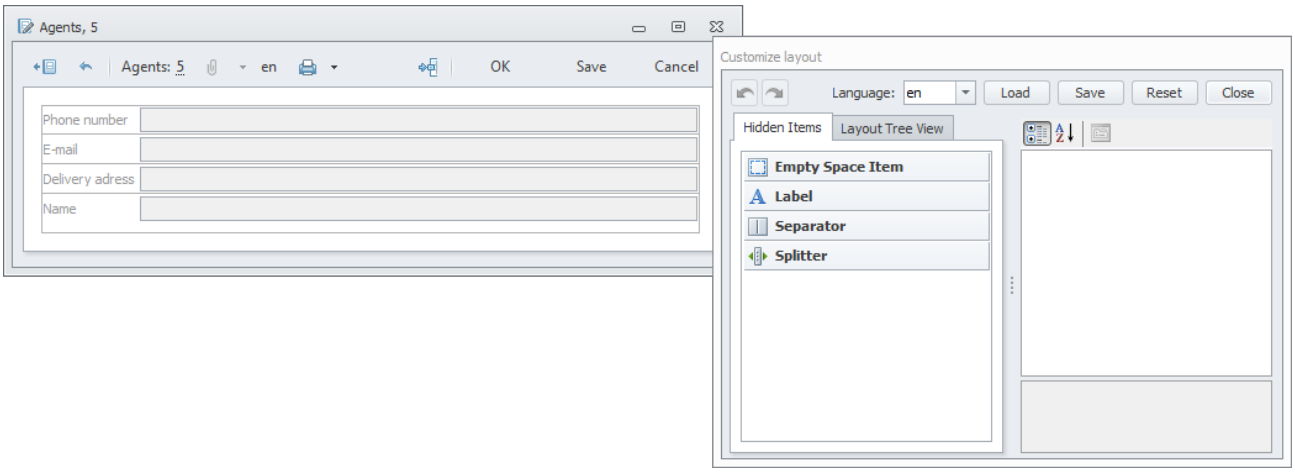
Open a dictionary via main menu and enter a new record:



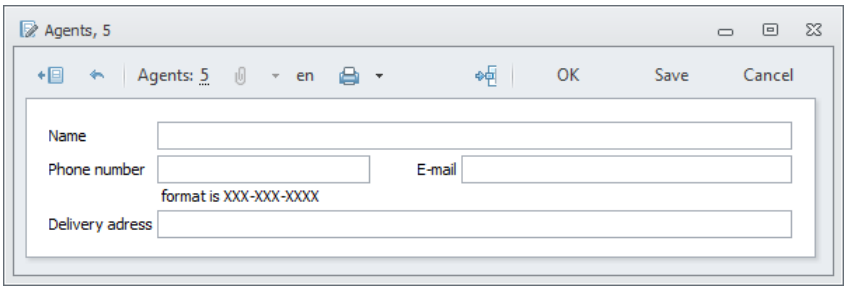
The opened standard editing form can be modified if necessary:



The interface is set separately for each system language:



The editing tools allow dragging the elements in the form, arranging and separating them, adding new ones, modifying, and deleting them. Among interface elements available for placing in the form, there are fields for all the dictionary properties, additional controls created by editor script, as well as buttons for dictionary commands:



In the same way, a filter of records in the dictionary list form can be set.

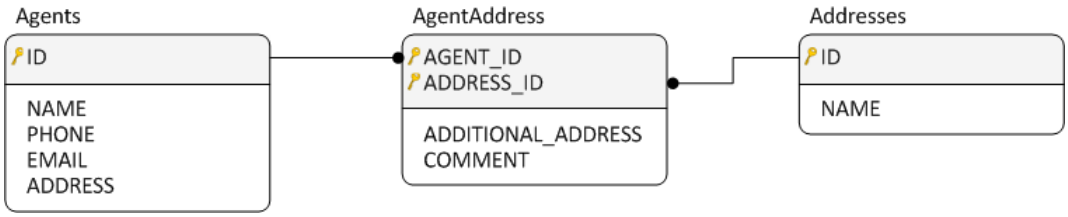
The columns displayed in the list form can be set with Select columns tool – button on the toolbar (hot key F2).

How to make link tables

Let us assume that we have a task to implement an option to add several additional addresses for the records of previously created Agents dictionary. Moreover, firstly, we do not know in advance how many additional addresses can be required in each particular case: there can be several of them and can be none. Secondly, a situation is not eliminated that no matter what number of additional addresses we assumed sufficient, a need may arise for some record in even larger number.

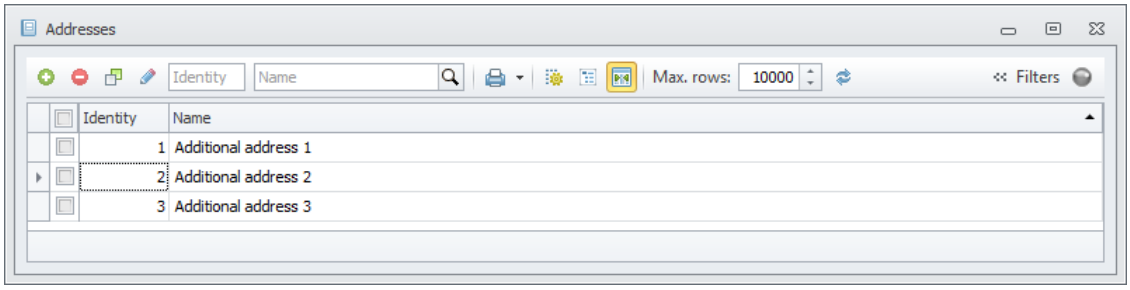
The task can be solved by adding several additional properties for storage of addresses in the Agents dictionary. If at any point of time they appear to be insufficient for some record, the application developer will be able to add new properties to the dictionary.

Or additional Addresses dictionary can be made, to store only the list of addresses like: *Address 1*, *Address 2*, *Address 3*... Link this dictionary using the link table to the Agents dictionary and store the addresses in the very link table:

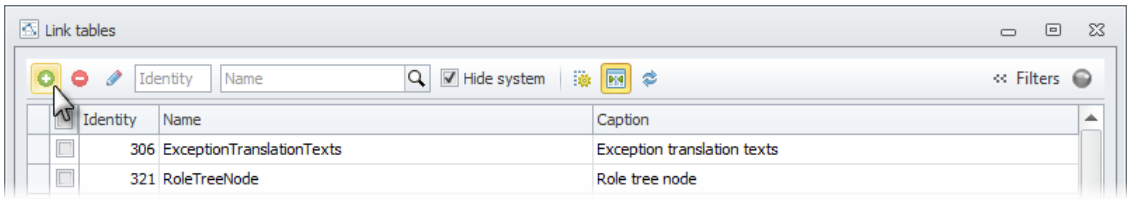


This option is better than the previous one because creation of new record in the Addresses dictionary will be sufficient in case of lack of addresses. Moreover, a common user having a permission to edit this dictionary will be able to do that.

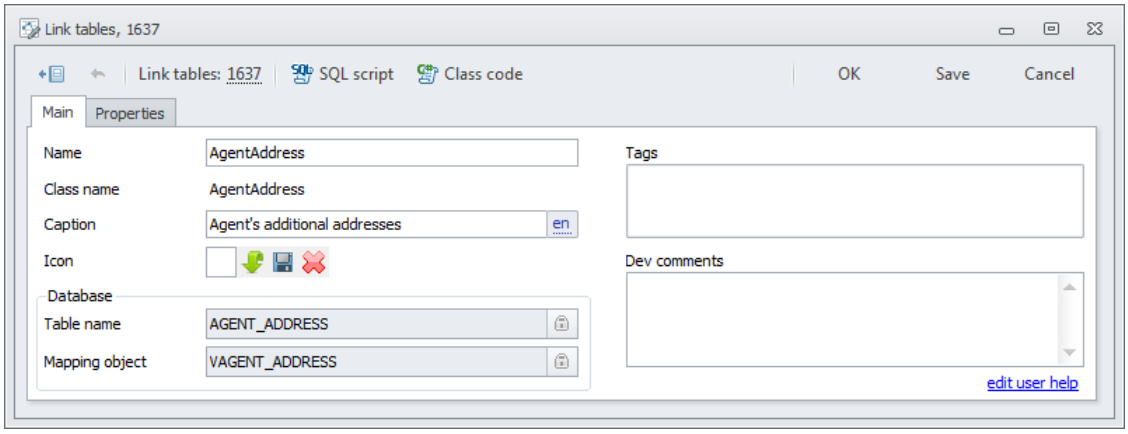
Creating and filling in of the Addresses dictionary. Let us dwell on three additional addresses at first:



In the dictionary of link tables (menu item Link tables), enter a new record:

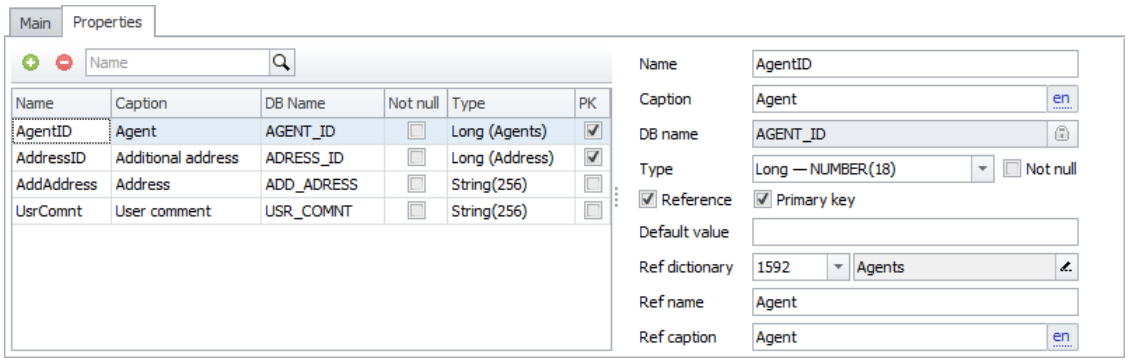


In the *Main* tab, set main parameters for the link table:



Name of the dictionary defines the name of its objects in the database. *Caption* description will be displayed in the screen forms.

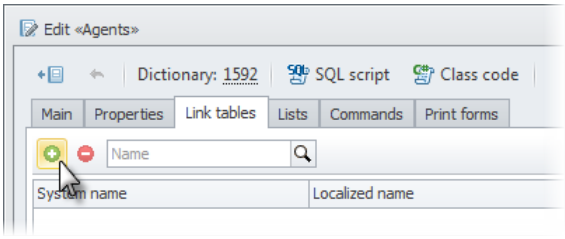
On Properties tab, set the properties of link table. Two of them we can do using the links to Agents and Addresses dictionaries:



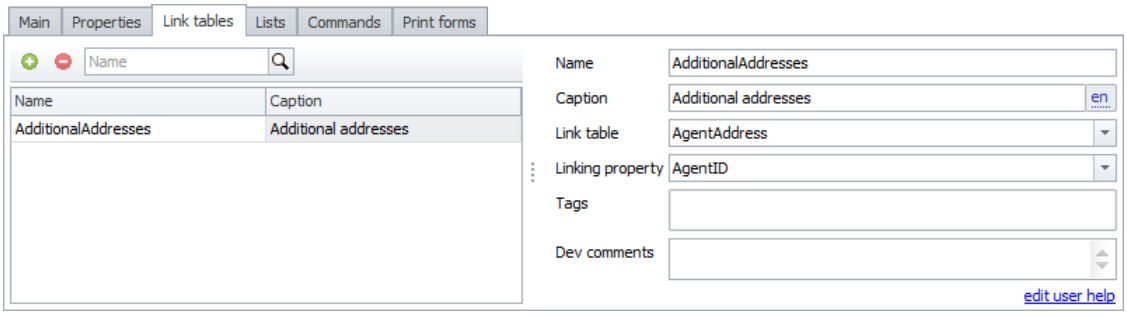
Save the link table, compile metadata and create the link table objects with the final step in DBMS in *Ultima* scheme.

The link table has no own form for editing, like for instance a dictionary. The records are added in it by the edit form of one or several of the dictionaries linked to it. In our case, adding of additional addresses for a record of Agents dictionary from its own edit form is logical and convenient.

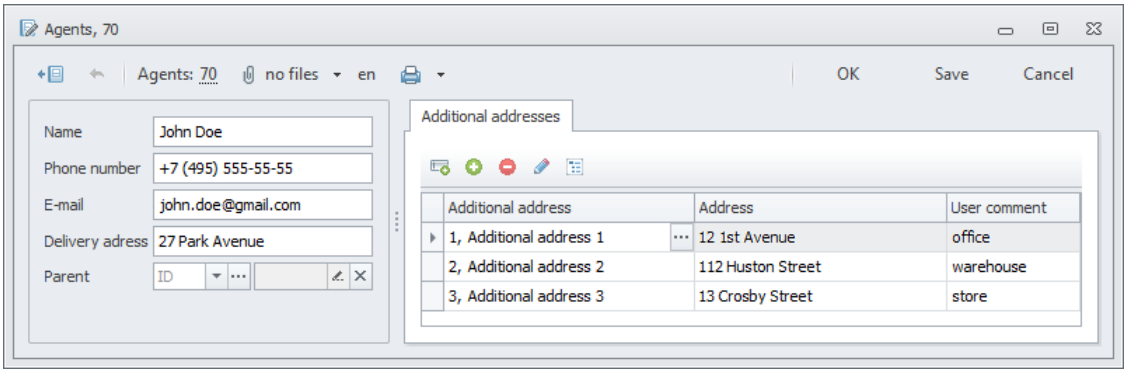
For that purpose, the created link table should be spelled out in the dictionary. Open the Agents dictionary in the Link tables tab and add new link table:



Choose the created link table as *Link table* property value, *Linking property* value is a property of link table, which the Agents dictionary refers to:



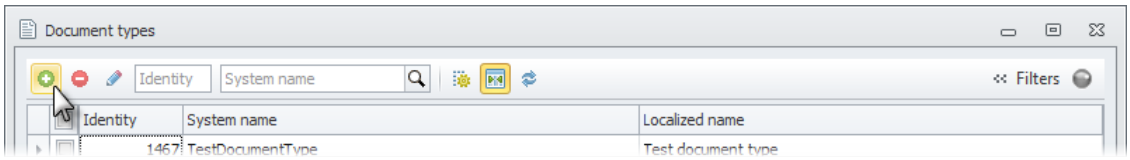
After that, the Additional addresses tab will appear in the dictionary edit form (the name coincides with the *Caption* property value, entered during adding of the link table into the dictionary), at which the link table data will be located, being linked to the dictionary record:



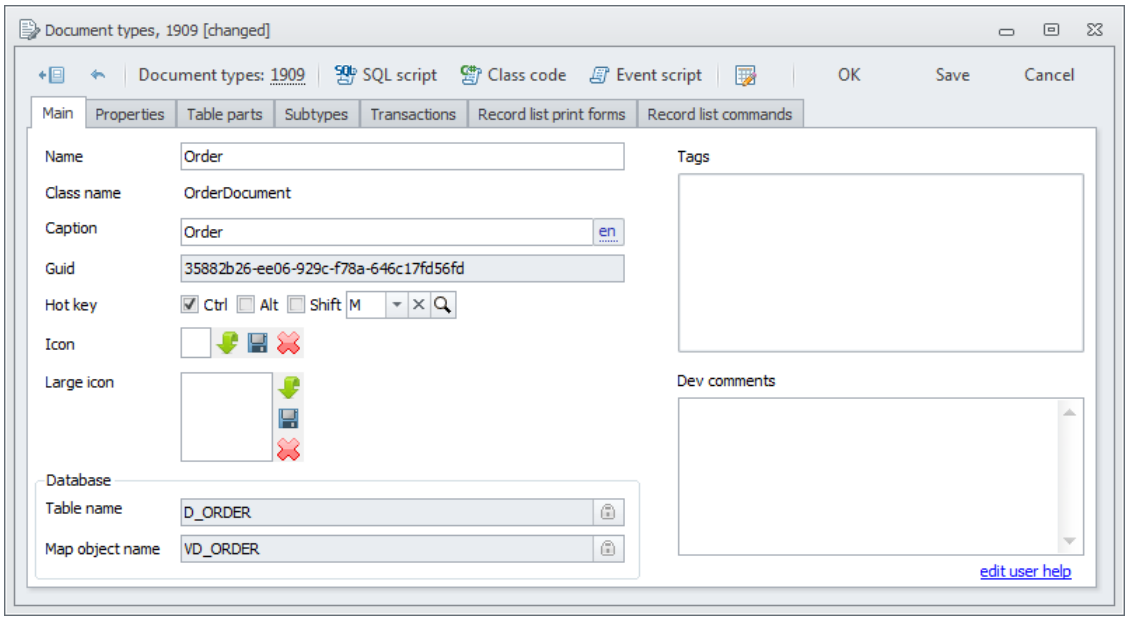
Let us proceed to more complex objects.

How to make a document

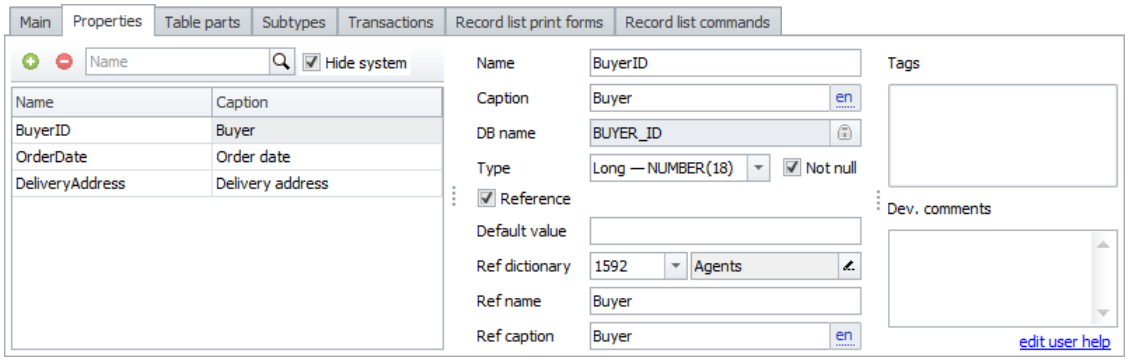
In the list form of document type dictionary (menu item Document types), enter a new record:



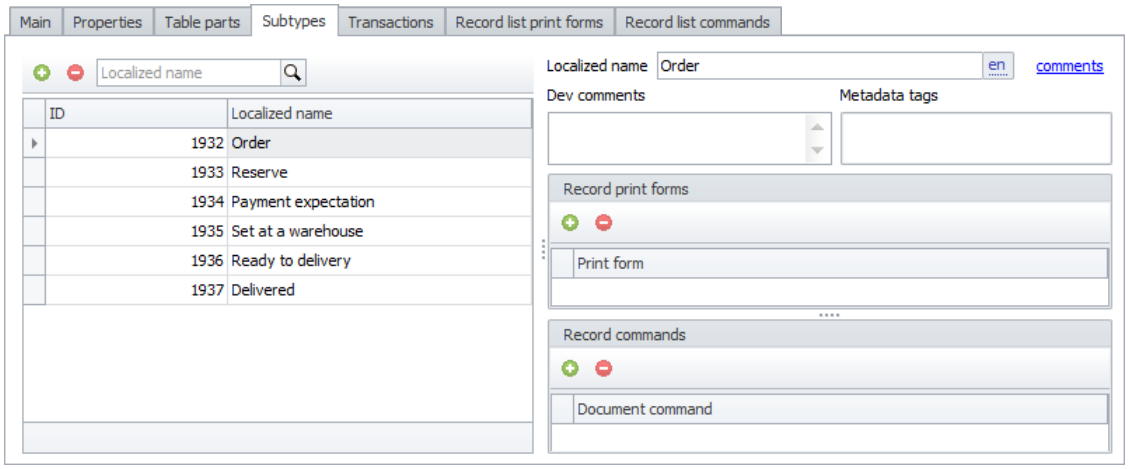
One the "Main" tab set the *Name* of document type, which also defines the names of its objects in the database and *Caption*:



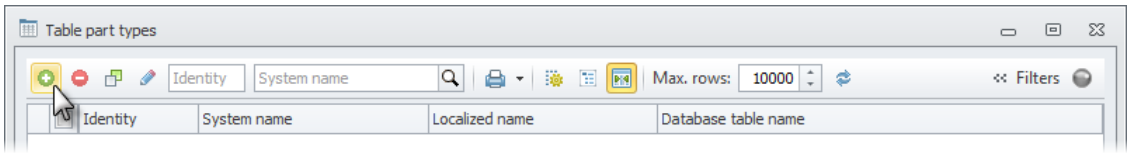
On "Properties" tab, set the properties of document type:



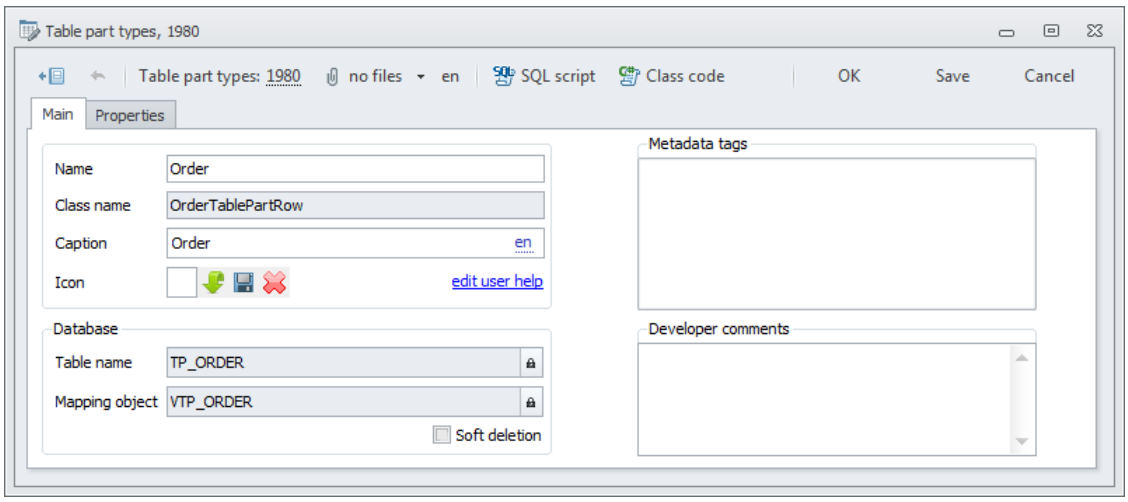
On "Subtypes" tab, create at least one document subtype:



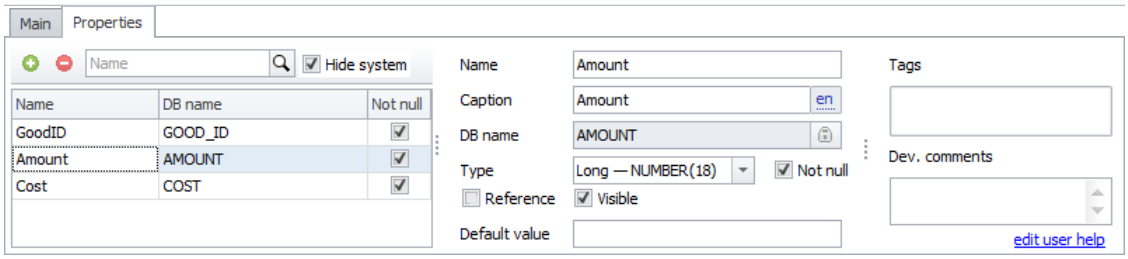
On "Table parts" tab, link the table part of the document, which should be preliminary created. For that purpose, in the list form of table parts (menu item Table part types), enter a new record:



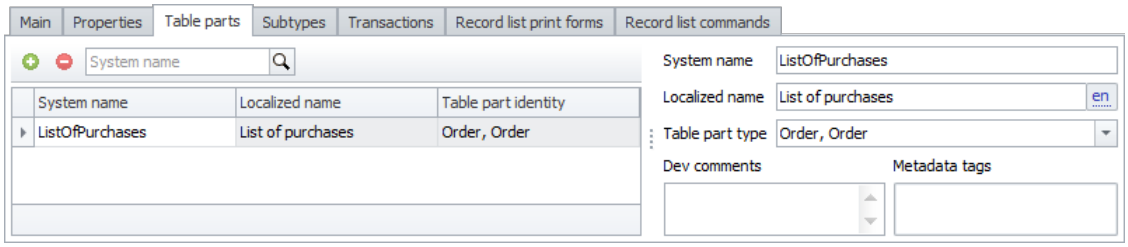
In the "Main" tab, set *Name* for the table part:



In the "Properties" tab, set its properties:



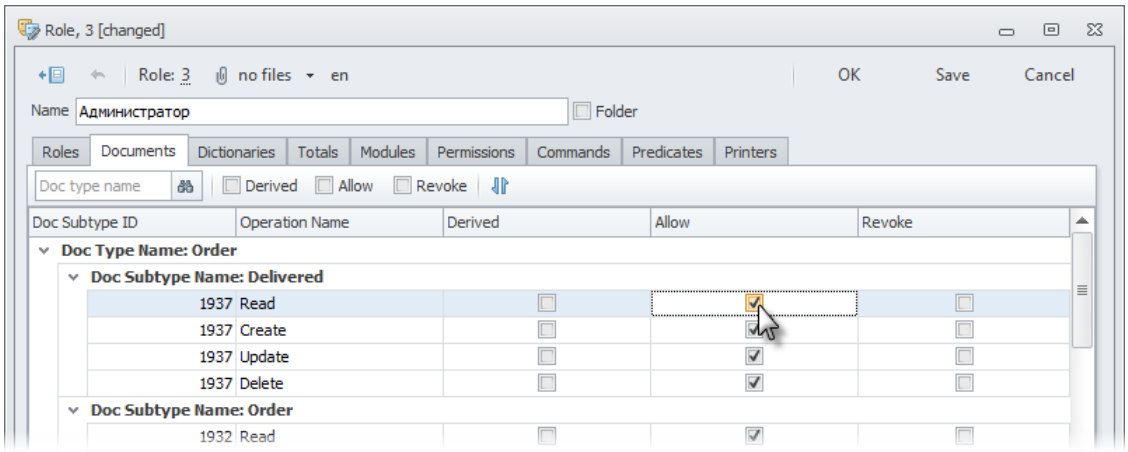
Save the table part. Now we can return to the document type edit form and link it:



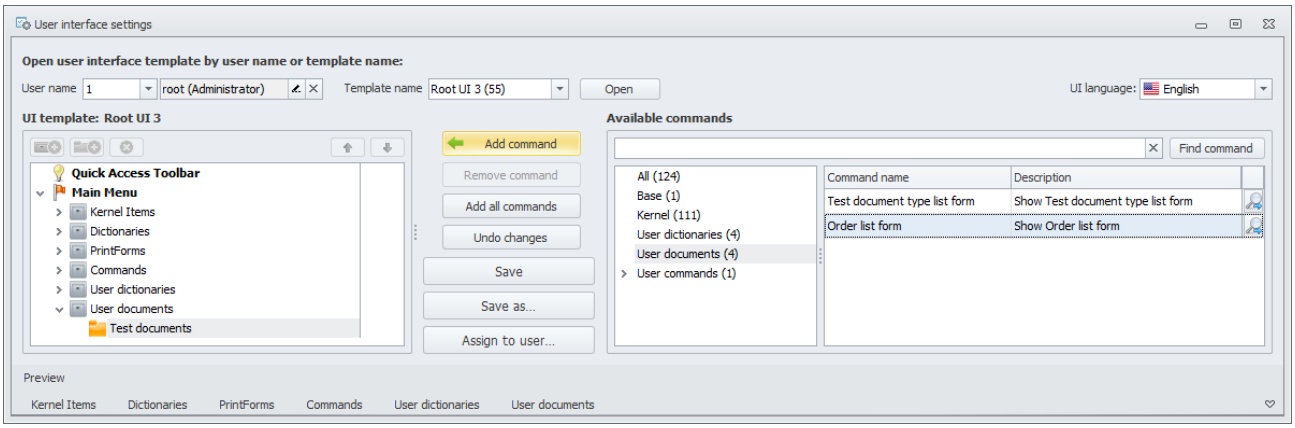
Now you can save the created document type, compile and reload the metadata.

The final stage will consist in creation of the objects for the table part and document type (just in this very sequence because the first one is used in the second one) in DBMS in *Ultima* scheme. It can be executed using SQL scripts generated with Ultimate AEGIS® application (button "SQL script" in the edit forms for table part and document type).

Now corresponding permissions should be granted to a user to handle the created document type. For that purpose, find the *Role* in the roles dictionary for the user, we want to grant these permissions to, and in the "Documents" tab, grant access to the subtypes of created document type (access is granted separately to each subtype):

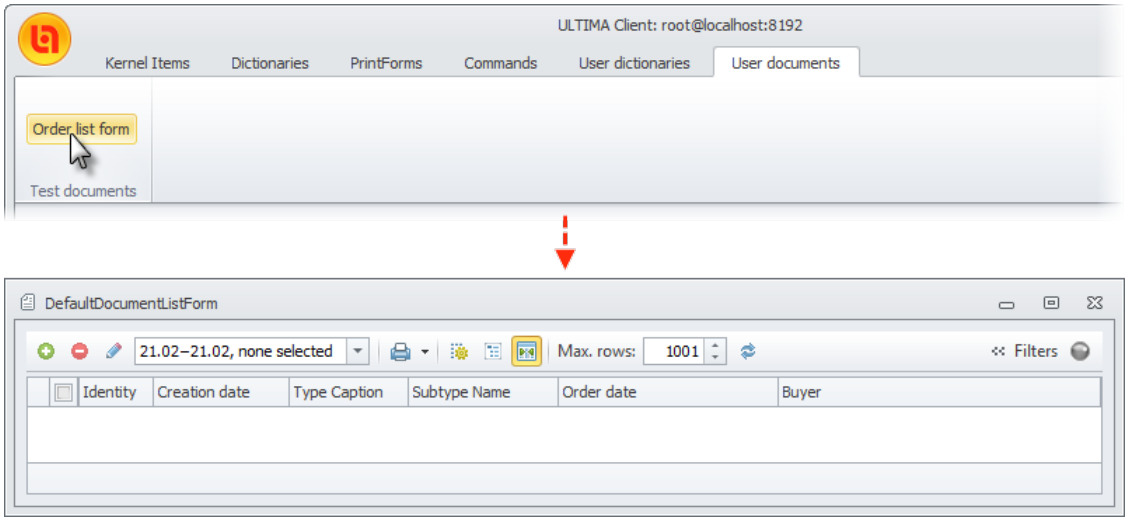


Besides, the command for opening of the list form of created document type should be added to the main menu using a form of user interface settings:

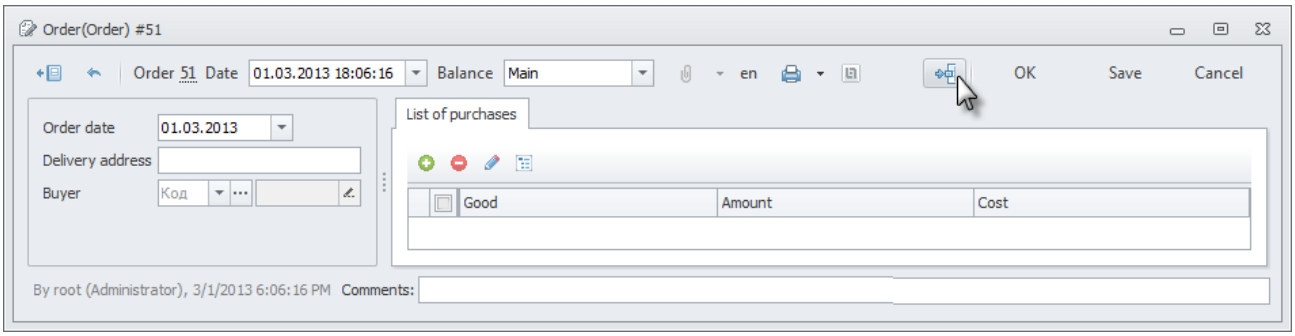


All types of documents including newly created one can be found among the objects of the group *User documents*.

After adding a document type to interface template and its saving, it becomes available in the main menu:

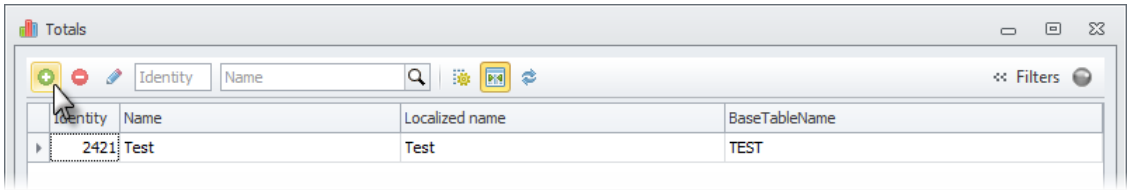


Modification of the standard screen form for editing of the documents of created type can be executed in the same way as for dictionaries (Editing of standard screen forms of the dictionary), and document editor scripts allows extending the functionality of the standard form:

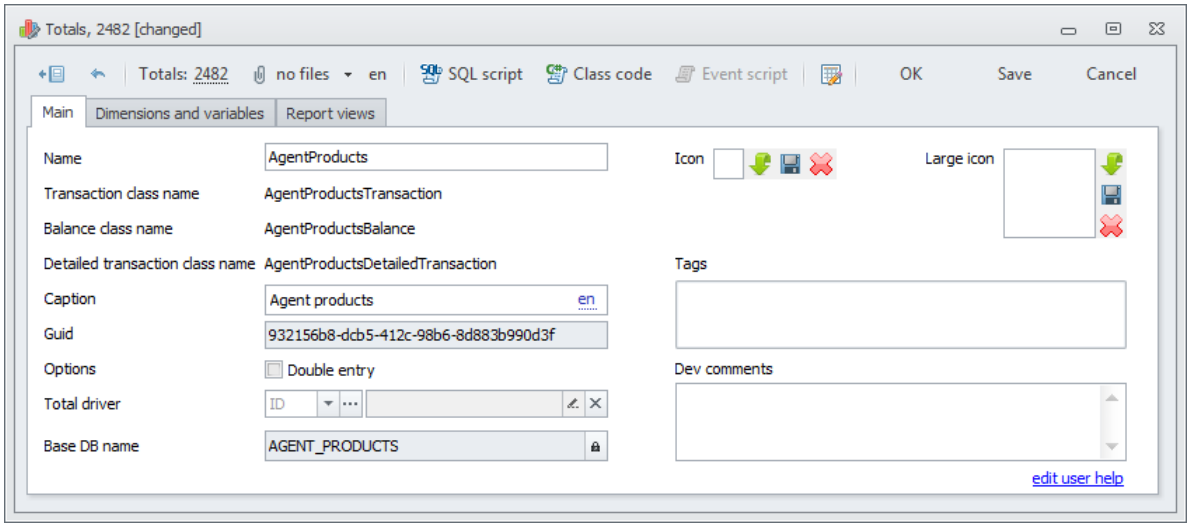


How to make total

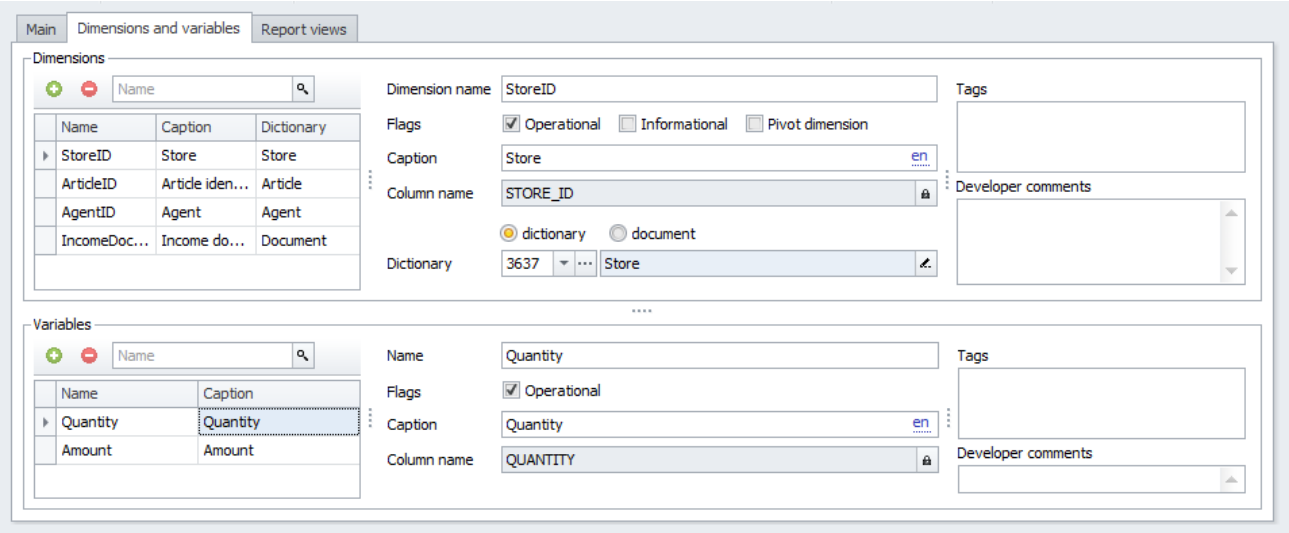
In the list forms of totals dictionaries (menu item Totals), enter a new record:



One the Main tab, set the total *Name*, which defines also the names of its objects in the database, and *Caption*:

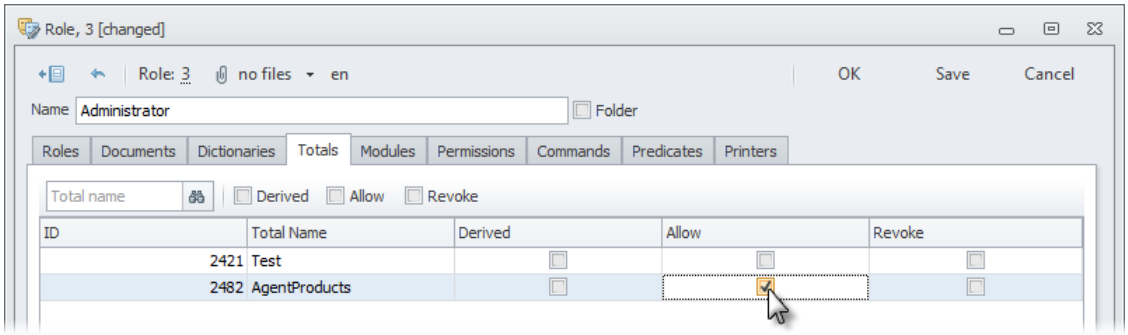


In the Properties tab, create dimensions and variables of the total:

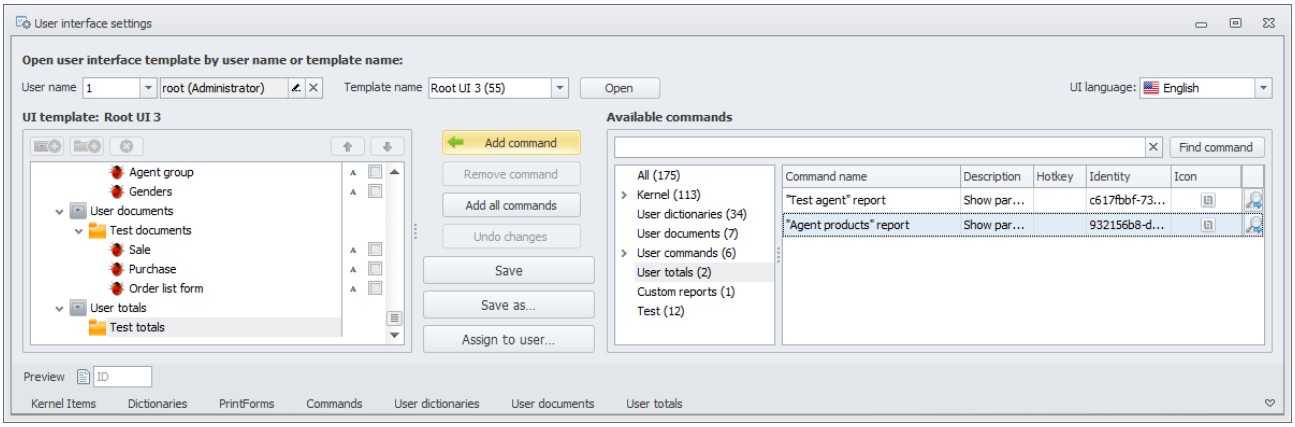


Save the created total, compile and reload metadata. The final step will consist in creation of objects total in DBMS in *Ultima* scheme.

Now corresponding permissions should be granted to a user for a possibility to view the reports on the created total. For that purpose, find the *Role* in the roles dictionary for the user, whom we want to grant these permissions to, and in the Totals tab, grant access to the created total:

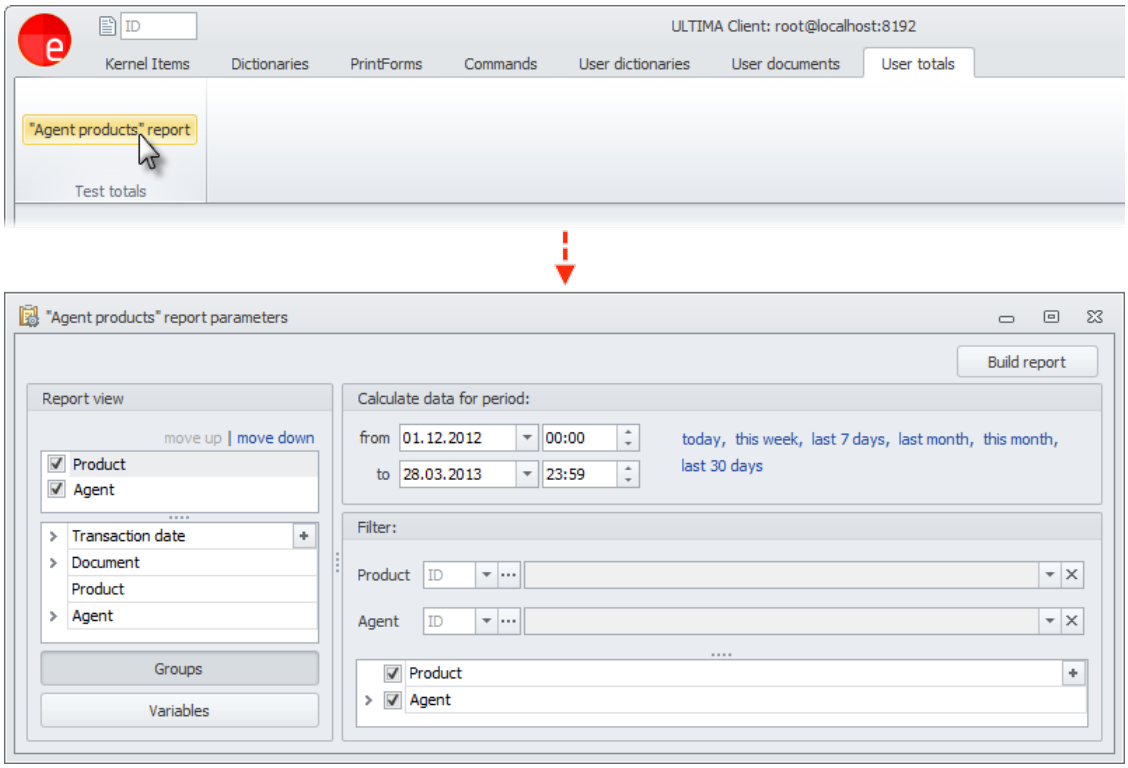


Besides, the command for opening of created total should be added to the main menu using a form of user interface settings:



All totals including newly created one can be found among the objects of the group *User totals*.

After adding a report on total to interface template and its saving, it becomes available in the main menu:



Scripts, handling of system events

System business logic is described in the form of programs or classes in C#, which start in case of particular events – saving of a document, dictionary record, etc. and executed at application server.

The class in C# itself, its resources for localization, a part generated by the system, etc. is called **script**. The system has built in script editor with IntelliSense technology support (see details on its use on MSDN site ⇨ eng/rus).

Additional parameters of script – system name, localized name for a user (which can be translated into other languages), binding to business object and other properties – are called its **main part**.

Therefore, entire business logic is described with pairs – script and its main part.

The application developer can implement the following types of scripts:

- user reports;
- dictionary record commands;
- dictionary list commands;
- document commands;
- documents list commands;
- handlers of dictionary record events;
- handlers of document events;
- transaction scripts;
- transaction validators;
- interfaces;
- print forms;
- services;
- tasks;

- providers of report columns;
- user commands;
- drivers of the totals;
- handlers of total events;
- general handler of dictionary record events;
- general handler of document events;
- web services and their DTO (Data Transfer Object);
- mobile services;
- mobile interfaces.

All scripts can be provisionally divided into two groups – executable in **protected** and **unprotected** modes. The scripts executable in the secure mode include:

- services;
- mobile services;
- web services;
- providers of report columns.

The key difference between these two groups is a method of permissions check:

- in insecure mode, the scripts are executed with disabled checks of user permissions. That is, if a user has no permissions to delete the document, he will be anyway able to delete it, having run some script, executable in insecure mode, which will delete the document. The user must surely have the permissions to execute (run) such a script;
- in the secure mode scripts are always executed with the rights of the user running them.

While calling a script, executable in the secure mode (e.g. service) from the script, executable in insecure mode, the permissions check will not be carried out either.

There are checks executed even in the unprotected mode (for example, when reposting documents, which are within an accounting period, the current user is verified if he is allowed to execute this operation). The only way to execute such operation is to give the user the corresponding permission.

The scripts, which the user can run from the client application by direct selection, are called *interactive*. These scripts can be of the following types:

- dictionary record commands;
- dictionary list commands;
- document commands;
- documents list commands;
- user commands;
- print forms.

The scripts of interactive group have a mechanism to display *forms of parameters* and *client actions*:

- parameters form is a form displayed in the client application for request of any data from the user. It can be described both in a declarative manner, just recollecting the types and names of parameters – the system will generate itself and show the form to the user, and own form can be created, to be shown before running the script;
- the mechanism of client actions allows transmitting the control commands from the application server into client application for performance of particular actions (e.g. the script has generated some file, which should be offered to user to save locally).

The types of scripts set the method and objectives of their use. Let us dwell on them in details.

Commands

Commands for the dictionary element:
Is called from the context menu of the screen form of the dictionary records list or dictionary record edit form.
The following is transferred to an entrance of the script:
- identifier of the chosen dictionary record;
- additional command parameters received from the parameters form;
- a collection of actions that should be performed on completion of the script work;

The applied developer has to explicitly specify for which dictionaries the command is available.

Commands for the list of dictionary records:
Is called from the screen form of the dictionary records list.
The following is transferred to an entrance of the script:
- identifiers of the chosen dictionary records;
- additional command parameters received from the parameters form;
- a collection of actions that should be performed on completion of the script work;

The applied developer has to explicitly specify for which dictionaries the command is available.

Commands for the document:
Is called from the context menu of the document list screen form or document edit form.
The following is transferred to an entrance of the script:
- identifier of the chosen document;
- additional command parameters received from the parameters form;
- a collection of actions that should be performed on completion of the script work;

The applied developer has to explicitly specify for which subtypes of the documents the command is available. Its call by the user will be possible only if he has the corresponding rights on it, and the command is available to the edited document subtype.

Commands for the list of documents:
Is called from the document list screen forms.
The following is transferred to an entrance of the script:
- identifiers of the chosen documents received from the parameters form;
- additional command parameters;
- a collection of actions that should be performed on completion of the script work;

The applied developer has to explicitly specify for which types of the documents the command is available. Its call by the user will be possible only if he has the corresponding rights on it, and the command is available for types of the chosen documents in the list.

User commands:
Is called from the command list of the main menu.
The following is transferred to an entrance of the script:
- additional command parameters received from the parameters form;
- a collection of actions that should be performed on completion of the script work;

Usually scripts of this type perform general tasks, such as:
- to restore a deleted document;
- to generate a new password for the user, etc.

Print forms

The print forms are used for printout of information or saving it into the file.

The following is transferred at script input:

- Type of the dictionary (optional);
- ID (or array of IDs) of printed record (document or dictionary);
- additional parameters of the command;

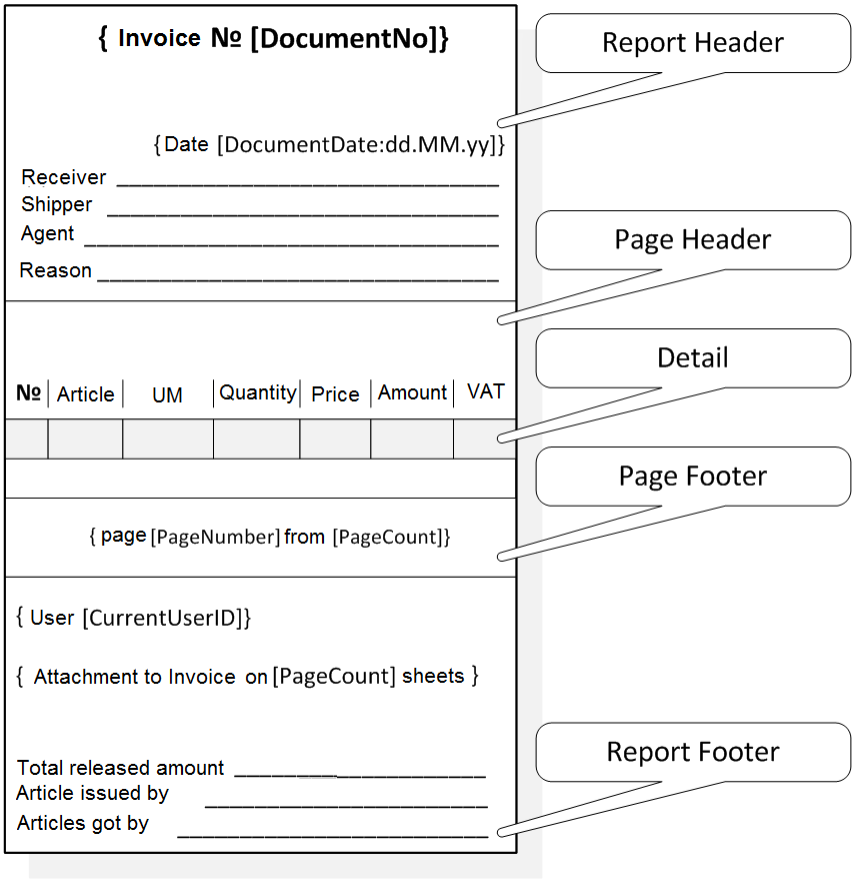
The script prepares data for filling in a template.

The main part of a printing form consists of one template at least. The template is processed by means of "engine" – a library that receives input data from the script, generates interface of the printing form on the template. Currently XtraReports is the engine.

The system allows export in the following formats:

- export into PDF;
- export into Word;
- export into Excel;
- export into HTML.

Templates of print form for XtraReports are created by means of graphic interface integrated into the system. An example of XtraReports template is given below:



Scripts of dictionaries

For each dictionary, one handler can be created, executable in a number of events, which may occur to dictionary records.

- The handler in case of creation of dictionary record *BeforeCreate*:
  Called in case of creation of a dictionary record.
  → A createddictionary record and parameters of its creation are delivered at input.
  The handler of this event is used generally for automatic fill-in of the fields of created dictionary record.

- The handler of initiation of dictionary record *AfterLoad*:
  Called after opening a dictionary record, but is not called in case of creation of new record.
  → The opened dictionary record and flag, defining if the internal objects of the dictionary record will be loaded, are delivered at input.
  Using the handler of this event, e.g. additional parameters can be loaded into the dictionary record.

- The handler before saving the dictionary record *BeforeSave*:
  called in case of saving of created/edited dictionary record.
  → The saved dictionary record is delivered at input.
  Using the handler of this event, e. g., one can check the data and, if necessary, modify the data before saving.
  ↠ After successful execution of the handler, the dictionary record will be saved, and the kernel will call the handler after saving; in case of failed execution, an error will be returned.

- The handler in case of failed saving of the dictionary record *SaveFailed*:
  Called in case of throwing an exception during operation for saving of the dictionary record.
  → The saved dictionary record and exception thrown during saving are delivered at input.

- The handler after saving the dictionary record *AfterSave*:
  called after successful execution of the handler before saving and directly after saving of the dictionary record but before transaction commitment..
  → The saved dictionary record is delivered at input.
  Using the handler of this event, saving can be canceled, having thrown an exception, and modifications made to the dictionary record can be canceled (modifications cannot be made to already saved record). The handler of this event can be also used for performance of automatic actions, which must be completed only after correct saving of all data.

- The handler before deleting the dictionary record *BeforeDelete*:
  called before deletion of the dictionary record.
  → ID of deleted dictionary record is delivered at input;
  Deletion can be canceled using this handler, having thrown an exception.

- The handler in case of failed deletion of the dictionary record *DeleteFailed*:
  called in case of throwing an exception during operation for deletion of the dictionary record.
  → ID of deleted dictionary record and exception thrown during deletion are delivered at input.

- The handler after deletion of the dictionary record *AfterDelete*:
  Called after successful execution of the handler before deletion as well as after successful deletion of the dictionary record (the dictionary records do not exist any more), but before transaction commitment.
  → ID of deleted dictionary record is delivered at input;
  Deletion can be canceled using this handler, having thrown an exception.

Besides, the application developer can modify the general handler of dictionary events - has the same interface but is called at any events of dictionary record of any type.

The sequence of calls of dictionary events handlers:

1. In case of saving:
   - general handler before saving;
   - handler before saving;
   - saving modifications of the record in DB;
   - general handler after saving;
   - handler after saving;

   If an error has occurred at any stages of saving, a handler will be called for exceptional situation *SaveFailed*.
2. In case of deletion:
   - general handler before deletion;
   - handler before deletion;
   - deletion of the record in DB;
   - general handler after deletion;
   - handler after deletion;

   If an error has occurred at any stages of deletion, a handler will be called for exceptional situation *DeleteFailed*.
3. In case of creation:
   - creation of the object of corresponding class;
   - receipt of new ID;
   - general handler during creation;
   - handler during creation;
4. In case of record loading:
   - loading of the record from DB and initiation of the object;
   - general handler of record initiation;
   - handler of record initiation.

Scripts of documents

For each document (type), one handler can be created, executable in a number of events, which may occur to the document.

The handler in case of creation of document *BeforeCreate*:
called in case of creation of a document.
The created document and parameters of its creation are delivered at input.
The handler of this event is used generally for automatic fill in of the fields of the head of created document.

The handler of document initiation *AfterLoad:*
called after opening a document but is not called at creation of new document.
The opened document and flag, defining if the internal objects of the document will be loaded, are delivered at input.
Using the handler of this event, e.g. additional parameters can be loaded into the document.

The handler before saving the document *BeforeSave*:
called in case of saving of created/edited document.
The saved document is delivered at handler input.
Using the handler of this event, the document data can be checked and, if necessary, modified before saving, for instance:

- check if the credit of recipient is used up;
- check if the shipped product is in storehouse and, if necessary, replace it with the same product from another storehouse, etc.

After successful execution of the handler, the document will be saved, and the kernel will call the transaction processor, or, in case of failed execution, an error will be returned.

Transaction scripts:
called with the kernel after successful execution of the handler before saving of the document. The transaction scripts are associated with document subtypes (they may be several for each document subtype) and executed at saving of the document of corresponding subtype.
The document, processed by the previous handler, collection of pairs of transactions (for balance totals) and collection of transactions (for non-balance totals) are delivered at input.
The handler creates the transactions booked by the kernel as totals.
As a result of successful execution, handler returns the array of transactions, and the kernel will call the handler after document saving, or, in case of failed execution, an error will be returned.

The handler in case of failed saving of the document *SaveFailed*:
called in case of throwing an exception during operation for document saving.
The saved document and exception thrown during saving are delivered at input.

The handler after document saving *AfterSave*:
called after successful execution of transaction processor (saving of transactions by the kernel into the database) and after direct saving of the document but before transaction commitment.
The document, processed by the previous handler, is delivered at input.
Using the handler of this event, saving can be canceled, having thrown an exception, and modifications made to the document can be canceled (modifications cannot be made to already saved document).
The handler is also used for automatic execution of actions, which must be performed only after correct saving of all data related to the document. For instance, upon processing of the order received from internet shop by the manager, an e-mail should be sent to the buyer with notification that their order awaits transfer to the delivery service. It could be executed using the handler before document saving but a situation is actually possible when after correct completion of its work, an error will be returned already by the transaction processor, followed with rollback of all modifications made to the system, and the order will never be saved.

The handler of generation of document description (value of the field Description) *GenerateDescription*:
called after document saving but before the handler after saving.
The document and description value are delivered at input;
Using the handler of this event, document description can be modified, which is generated by default according to the template *{DocumentType}({DocumentSubtype}) #{ID} {TRANSACTION_DATE}*.

The handler before deletion of the document *BeforeDelete*:
called before deletion of the document.
ID of deleted document is delivered at input;
Deletion can be canceled using this handler, having thrown an exception.

The handler in case of failed deletion of the document *DeleteFailed*:
called in case of throwing an exception during operation of document deletion.
ID of deleted document and exception thrown during deletion are delivered at input.

The handler after document deletion *AfterDelete*:
called after successful execution of the handler before deletion as well as after successful deletion of the document (the document does not exist any more), but before transaction commitment.
ID of deleted document is delivered at input;
Deletion can be canceled using this handler, having thrown an exception.

Besides, the application developer can modify the general handler of document events - has the same interface but is called at any document events (except for generation of its description) of any type.

The sequence of calls of document events handler is as follows:

1. In case of saving:
   - general handler before saving;
   - handler before saving;
   - transaction scripts;
   - saving of the head and data of table parts in DB;
   - saving of transactions in totals;
   - handler of document description generation;
   - general handler after saving;
   - handler after saving;

   If an error has occurred at any stages of saving, a handler will be called for exceptional situation *SaveFailed*.
2. In case of deletion:
   - general handler before deletion;
   - handler before deletion;
   - deletion of the document in DB;
   - deletion of transactions from totals;
   - general handler after deletion;
   - handler after deletion;

   If an error has occurred at any stages of deletion, a handler will be called for exceptional situation *DeleteFailed*.
3. In case of creation:
   - creation of the object of corresponding class;
   - receipt of new ID;
   - general handler during creation;
   - handler during creation;
4. In case of document loading:
   - loading of the document from DB and initiation of the object;
   - general handler of document initiation;
   - handler of document initiation.

Services and interfaces

Services

The services are designed to store frequently used functionality. Each service implements one of described *interfaces*, which allows calling corresponding methods both from those executed on the server and from the client application. Once having written the service for solving of typical task, you can resort to its functionality, while importing its implemented *interface*.

Interfaces

To describe the available set of functions for the *service* its interface should be also described. The interfaces comprise a part of metadata and compiled together with the classes of dictionaries and documents. Therefore, the compiled classes announced in the interface are available both on the server and in client application.

Mobile services

Implementation of services for mobile devices having Xamarin platform support. The mobile service must be implemented by one mobile interface.

Mobile interfaces
Implementation of interfaces for mobile devices having Xamarin platform support. In view of limitations of software platform, they are separated from general interfaces. The mobile devices are compiled into a separate library, which can be used in mobile application.

Web services
Allow being integrated with other systems on http, using the SOAP protocol or through REST services. Implemented using message-based design.

Totals

Drivers of the total
Called according to schedule by the mechanism for calculation of the kernel totals.
The drivers of the total are applied during calculation of variables of analytical tables of the totals. The example of driver operation is given in the section Documents.

Transaction validators
Called in case of saving a document.
The validators are applied to check the validity of the transactions created by the transaction scripts.

User reports
There are two types of reports in the system:
- reports on the totals, which functionality is implemented completely with the kernel ;
- user reports.

The scripts of this type are called with the kernel during formation of a user report.
Applied to provide the user with an option to process and filter data, formulate only the request kernel .
As as result of its operation, the request kernel and list of report columns is returned.

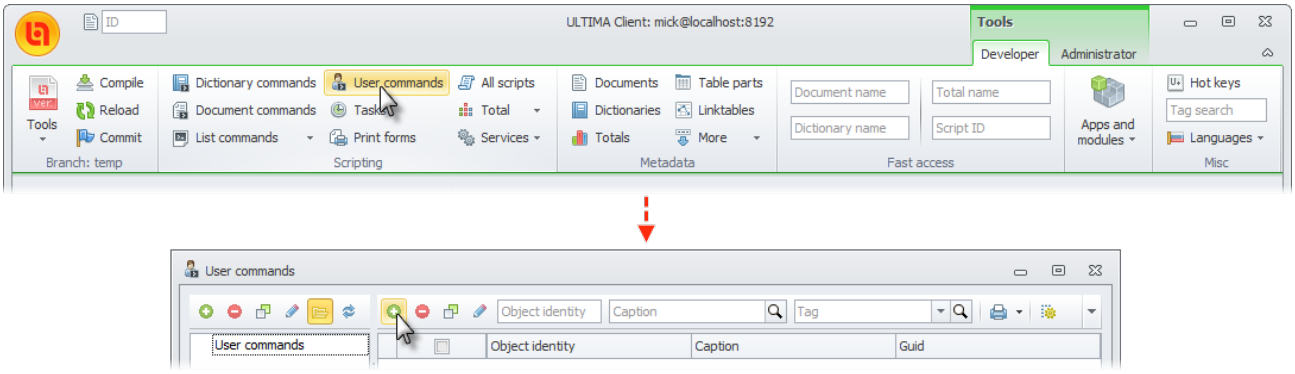
Column providers
Called by the kernel as part of report calculation.
The report grouping parameters are delivered at input.
During formation of standard or user reports, the situations may arise when the system faces the data structure, different from the standard one. In this case, it should be specified how to process them. The Column providers are applied for that purpose.
As a result of operation, the providers return the parts of requests used afterwards for formation of the report columns.

Creating simple commands

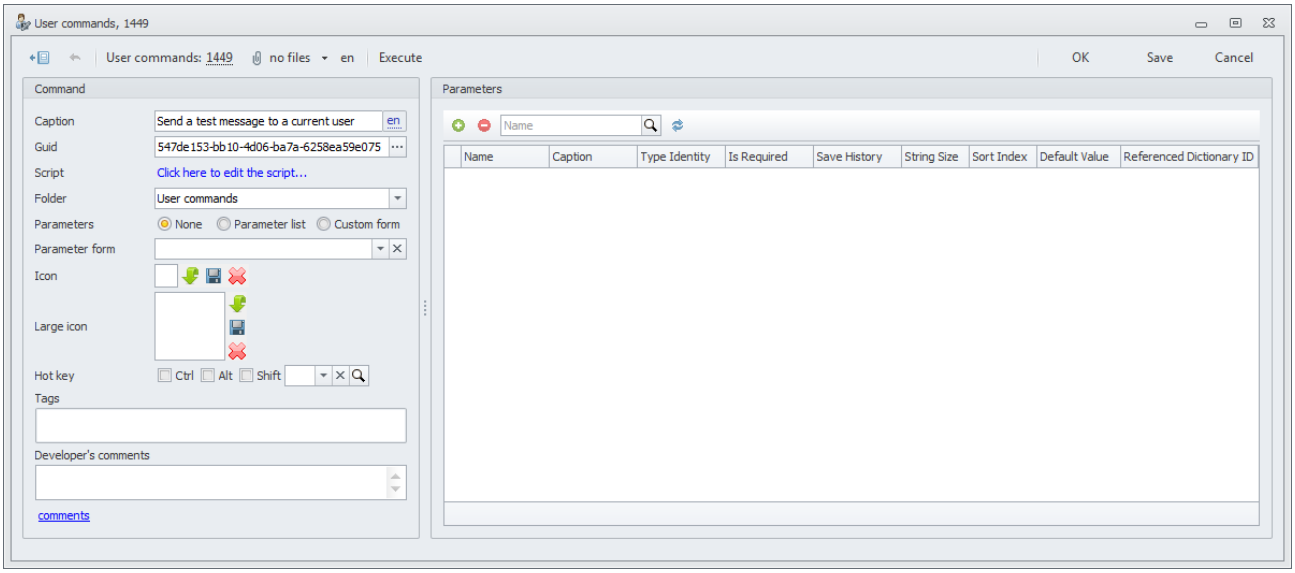
Let us explore how to create a simple command by the example of user command sending a message to the user executing it.

Creating a command

Create a new record in the dictionary of user commands:



Set *Caption* for new command and save the record:

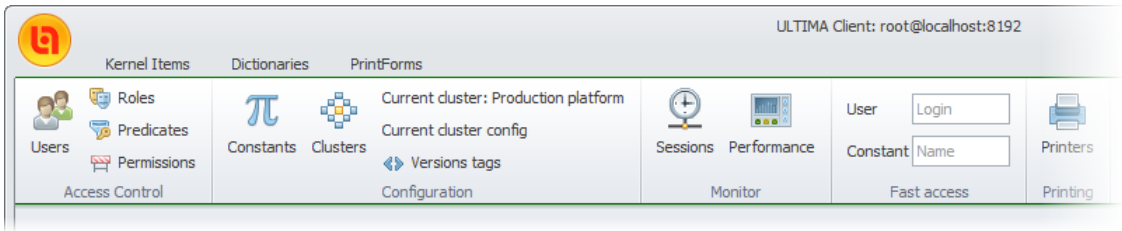


After saving the command, a script is created as well as selection of *Hot key* for command calling and *Parameters* become available.

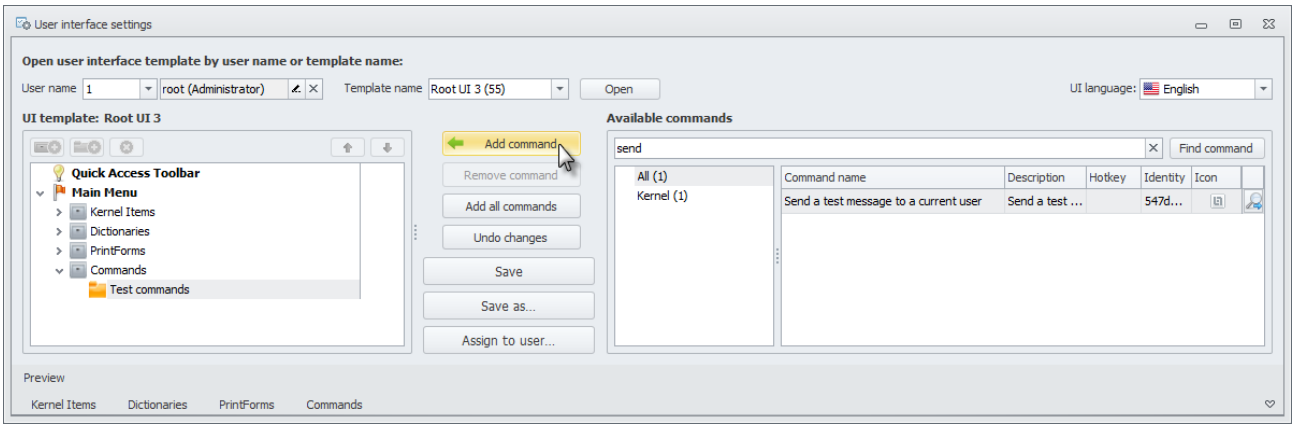
Adding a command to the interface

In order the user could execute a command, an option should be provided to call it.

In addition to combination of hot keys, the user dictionaries can be fetched via main menu:



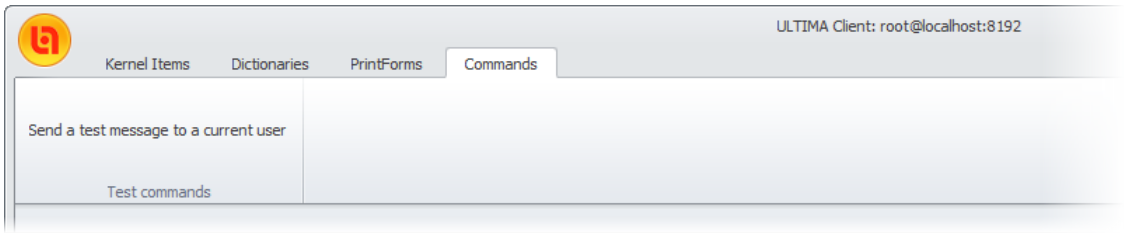
To add the created command to the main menu, use a form for setting of user interface (item *UI settings* of menu):



The interface template should be edited for the user, who requires provision with a possibility to call a command (in our case, it is *Administrator* with login *root*).

All user commands including newly created one can be found among the commands of group *Kernel*.

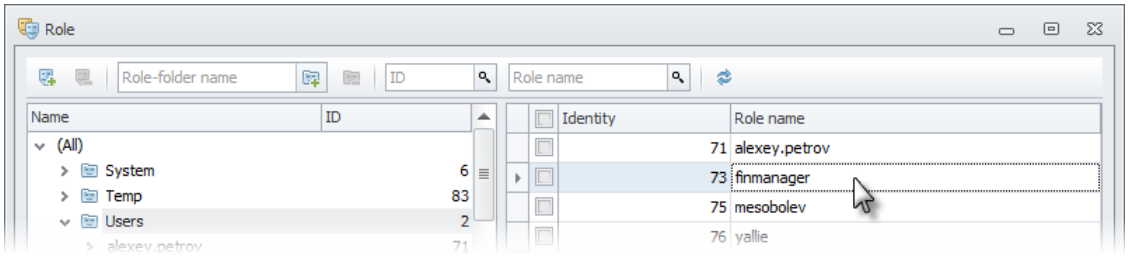
After adding a command to interface template and its saving, the command becomes available in the main menu:



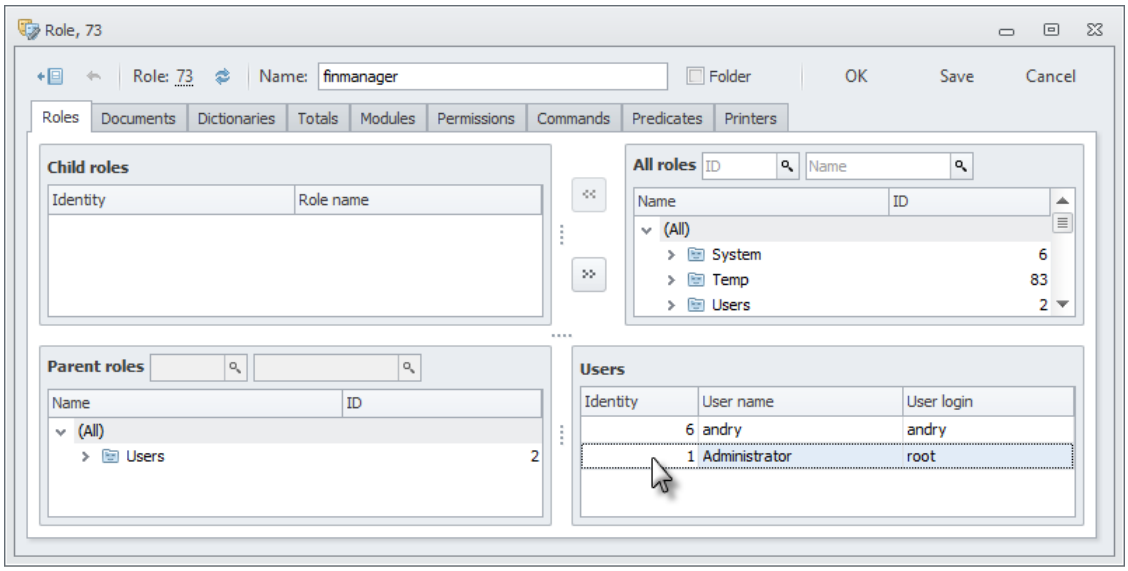
Issuing of permissions

In addition to the option to call a command, the user must have the permissions to its execution.

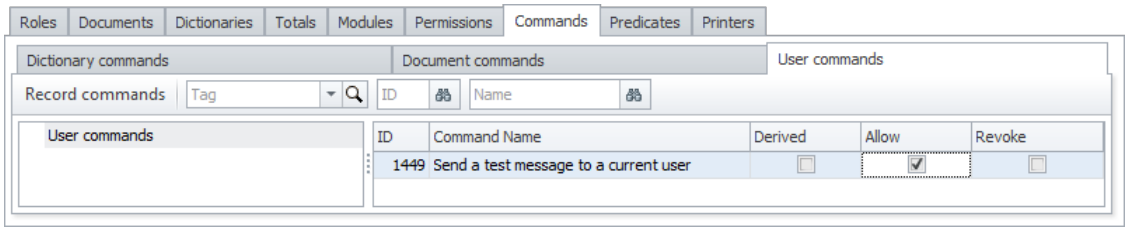
For that purpose, in the roles dictionary find a *Role* of the user, who should get a possibility to execute a command:



In the Roles tab, you can make sure that the edited role is assigned to the very user, whom we grant access to execute a command (in our case, it is *Administrator* with login *root*). Right there you can see, what other users are, who can be assigned with the role, and who will be also granted access to execute a command:



On subtab "User Commands" of tab "Commands", grant access *Allow* to execution of the set command:

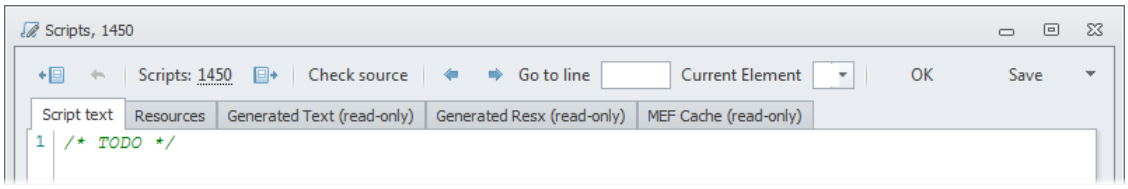


Now, after saving the modifications made to the role, *Administrator* has corresponding permissions to execute a command.

Editting a script

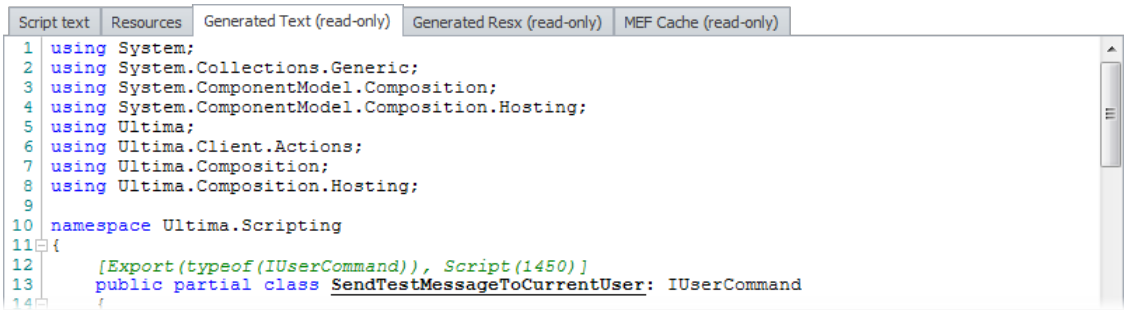
Let us return to created command.

Following the link *Click here to edit the script...* in the command edit form, you can open the form to edit its script:



Let us name the script class and, therefore, the script itself *SendTestMessageToCurrentUser*.

The scripts are inherited from the interface *IUserCommand*. It can be seen in the part of the script, which is generated automatically:

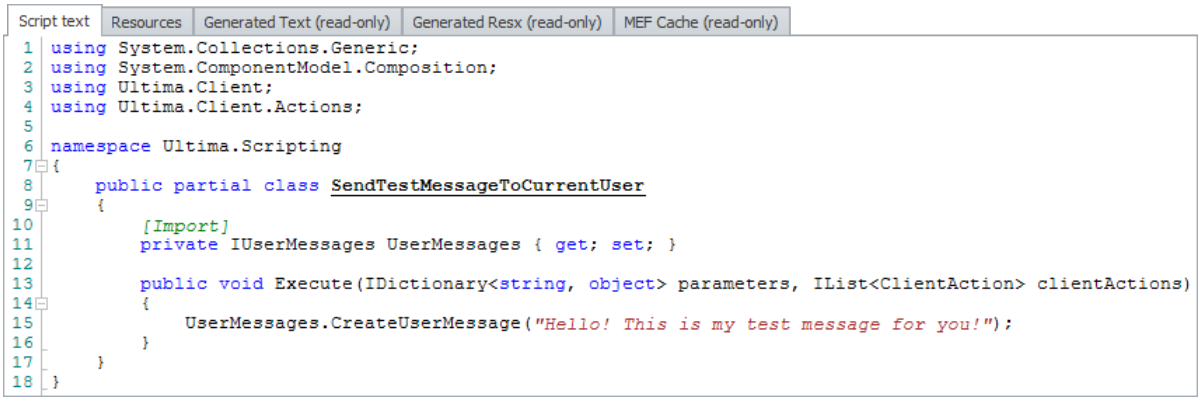


To send messages, we will require interface *IUserMessages*.

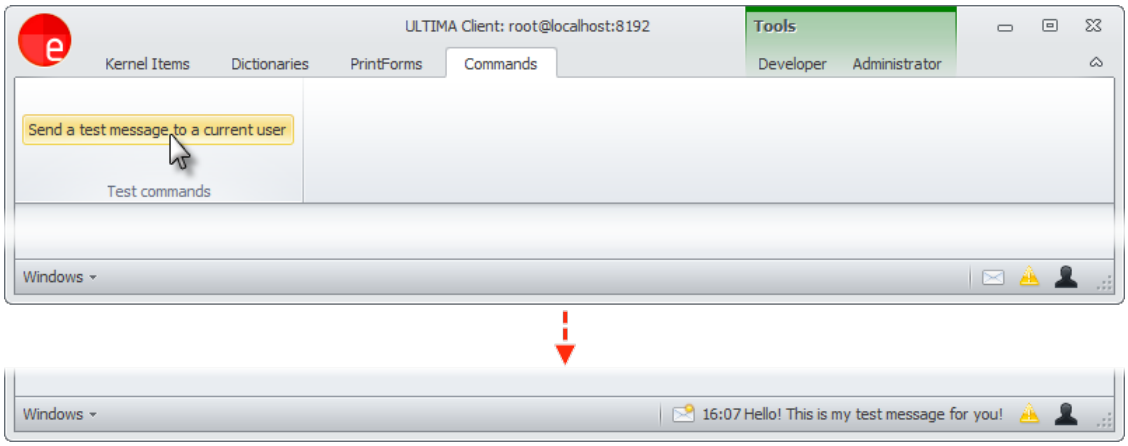
To use the services provided by the system, they should be imported. To do that, a property of the set type should be announced and it should be marked with attribute *[Import]*. Import is carried out using MEF, which details can be found in section Special managers:

```
[Import]
private IUserMessages UserMessages { get; set; }
```

Having imported the interface, it is possible to write the message to the user, realized the method *Execute* of the interface *IUserCommand*:



Check the script for errors (button *Check source* in the script toolbar) and save it. Now it is possible to execute the command:



Accessing data via LINQ

Let us make a task more complex, having considered a possibility to access data using LINQ queries at the same time. To do that, put the name of current user in the message.

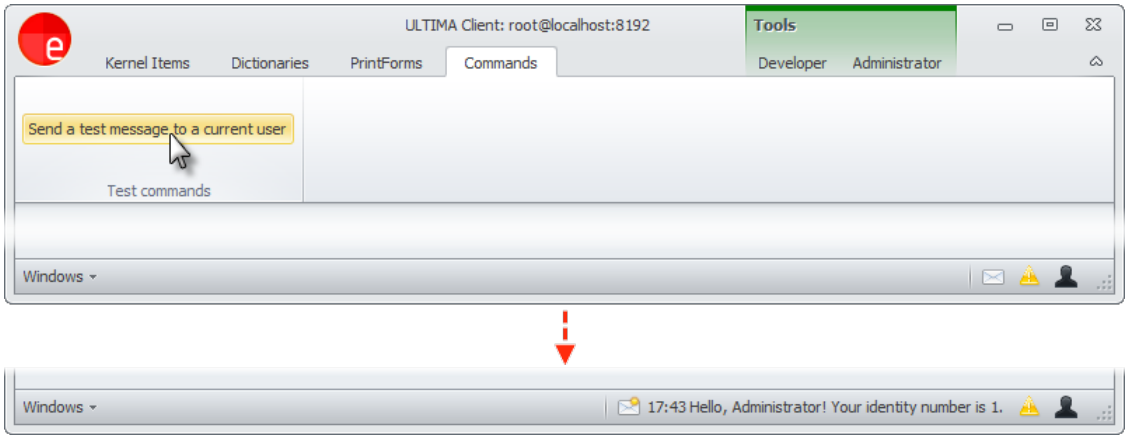
Access to each system object can be obtained through its corresponding class using *DataContext* class. To do that import the *ITableSource* interface.

Get *IUserManager* interface to obtain ID of the current user. Actually, it would be easier to obtain a user name using it too but we have a task to use LINQ query:

Script text | Resources | Generated Text (read-only) | Generated Resx (read-only) | MEF Cache (read-only)

```
using System.Collections.Generic;
using System.ComponentModel.Composition;
using System.Linq;
using Ultima.Client;
using Ultima.Client.Actions;
using Ultima.Linq;
using Ultima.Metadata;

namespace Ultima.Scripting
{
    public partial class SendTestMessageToCurrentUser
    {
        [Import]
        private ITableSource DataContext { get; set; }

        [Import]
        private IUserManager UserManager { get; set; }

        [Import]
        private IUserMessages UserMessages { get; set; }

        public void Execute(IDictionary<string, object> parameters, IList<ClientAction> clientActions)
        {
            // get current user ID
            var userId = UserManager.CurrentUserID;

            // get current user name using LINQ
            var userName = DataContext.GetTable<User>()
                .Where(u => u.ID == userId)
                .Select(u => u.Name)
                .Single();

            // send message for current user
            UserMessages.CreateUserMessage("Hello, {0}! Your identity number is {1}.", userName, userId);
        }
    }
}
```

Obtain user ID, request a name corresponding to this ID and send a message:



SQL queries

The same can be implemented with SQL queries, using *SqlService*:

Script text | Resources | Generated Text (read-only) | Generated Resx (read-only) | MEF Cache (read-only)

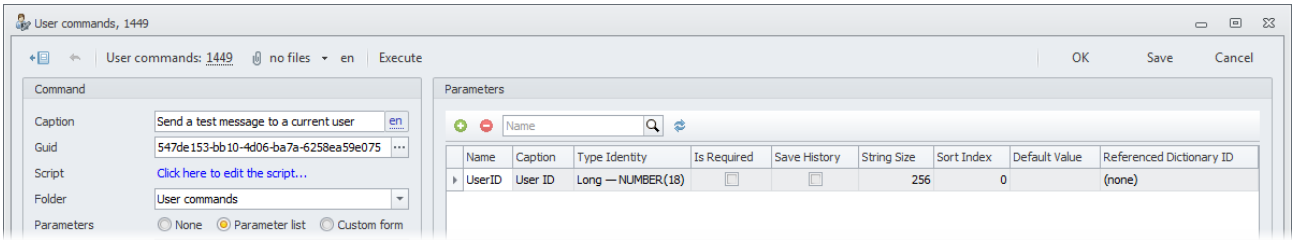
```
using System.Collections.Generic;
using System.ComponentModel.Composition;
using Ultima.Client;
using Ultima.Client.Actions;
using Ultima.Server.Data;

namespace Ultima.Scripting
{
    public partial class SendTestMessageToCurrentUser
    {
        [Import]
        private IUserManager UserManager { get; set; }

        [Import]
        private IUserMessages UserMessages { get; set; }

        public void Execute(IDictionary<string, object> parameters, IList<ClientAction> clientActions)
        {
            // get current user ID
            var userId = UserManager.CurrentUserID;

            var userName = string.Empty;

            // get current user name using UltimaDbManager
            using (var db = new UltimaDbManager())
            {
                var query = "select Name from KERNEL.USERS where ID = :vID";
                userName = db.SetCommand(query, db.Parameter("vID", userId)).ExecuteScalar<string>();
            }

            // send message for current user
            UserMessages.CreateUserMessage("Hello, {0}! Your identity number is {1}.", userName, userId);
        }
    }
}
```

Additional parameters query

If necessary it is possible to request the user, executing the command, to add the values of additional parameters and to use them further when performing a script. In this case, the command execution will be preceded by the opening of a form of additional parameters input. It is possible, for example, to remove the user name in the message instead of the current user, corresponding to the identifier added in the form of additional parameters.

Additional parameters of the command *Parameters* can be requested by means of the standard automatically generated form (flag *Parameters list*), previously adding them to a list titled *Parameters*. Or it is possible to request them by means of independently designed form (flag *Custom form*), in this case there is no need to add parameters to the *Parameters* list, however it is necessary to design a parameters form independently. Let's consider the first, simpler option.

In the form of command editing it is necessary to choose the *Parameter list* flag for the property *Parameters*, and to create necessary parameters in the list *Parameters* :



Each parameter has:

- *Name* — parameter name;
- *Caption* — name displaying in screen forms;
- *Type Identity* — parameter type (for more details see the section Data types);
- *Is Required* — flag, indicating whether the parameter is required to fill;

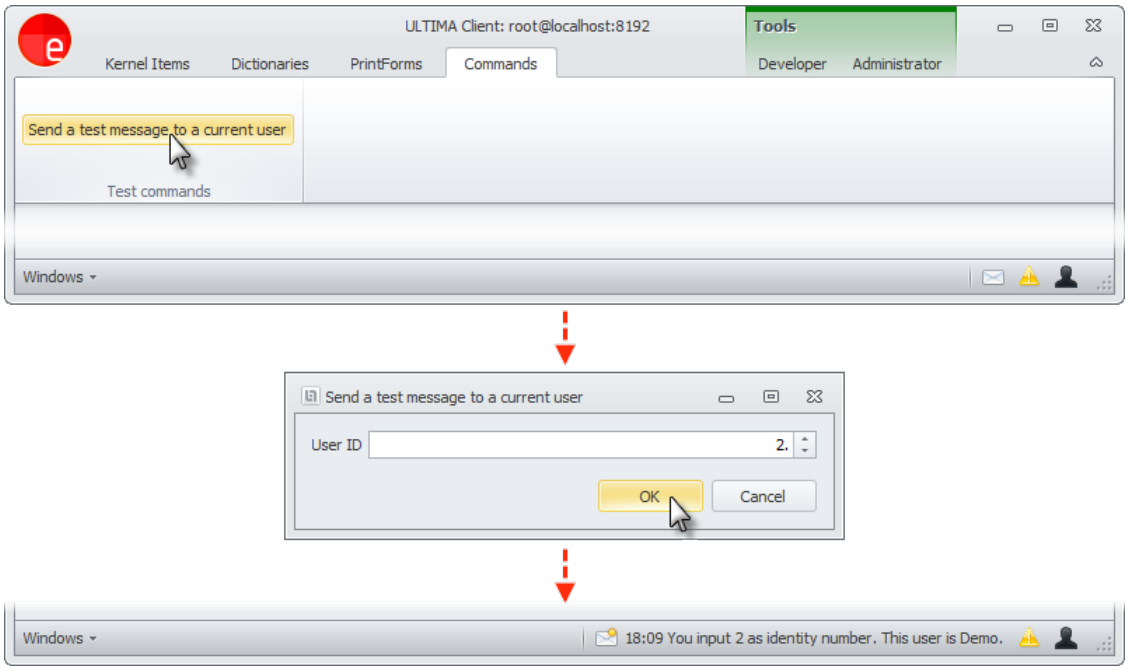
- *Save History* — flag, indicating the need to remember the last user-added value;
- *String Size* (available for data types *Text* and *String*) — limits the parameter size in specified value;
- *Sort index* — index, by which the parameters in a screen form will be sorted. As index values any integers can be used. Parameters will be ordered in the form from top to down in increasing order of the index;
- *Default Value* (is available for all data types except *Binary*) — parameter value by default which is used in the form of additional parameters. Values of parameters have to be constants (integers, lines), for dates it is also possible to use special values DateTime.Now and DateTime.Today;
- *Referenced Dictionary ID* (is available for data types *Long*) — dictionary ID (object), which the parameter is linked to.

Then you can pass to the script editing:

Script text | Resources | Generated Text (read-only) | Generated Resx (read-only) | MEF Cache (read-only)

```
using System.Collections.Generic;
using System.ComponentModel.Composition;
using Ultima.Client;
using Ultima.Client.Actions;
using Ultima.Server.Data;
using Ultima.Collections;

namespace Ultima.Scripting
{
    public partial class SendTestMessageToCurrentUser
    {
        [Import]
        private IUserManager UserManager { get; set; }

        [Import]
        private IUserMessages UserMessages { get; set; }

        public void Execute(IDictionary<string, object> parameters, IList<ClientAction> clientActions)
        {
            decimal dUserId = default(decimal);

            // try to get user ID from parameters
            if (!parameters.TryGetValue("UserID", out dUserId))
            {
                return;
            }

            var userName = "Unknown user";

            // get current user name using UltimaDbManager
            using (var db = new UltimaDbManager())
            {
                var query = "select Name from KERNEL.USERS where ID = :vID";
                userName = db.SetCommand(query, db.Parameter("vID", dUserId)).ExecuteScalar<string>();
            }

            // send message for current user
            UserMessages.CreateUserMessage("You input {0} as identity number. This user is {1}.", dUserId, userName);
        }
    }
}
```

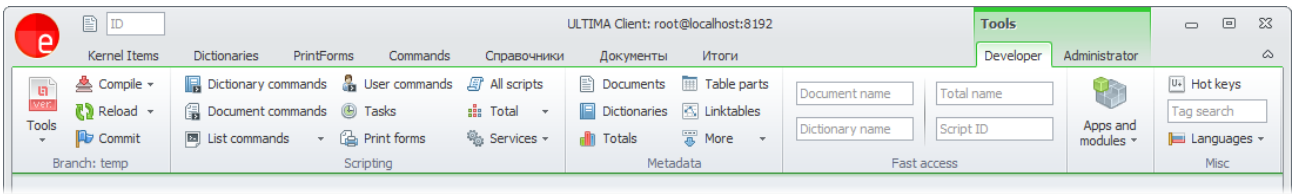
We get the user ID from a form of additional parameters. Control element *SpinEdit*, used in the parameters form generated by the Ultimate AEGIS® system, returns a value of type *decimal*. We request the user name corresponding to this id, using *SqlService*. Finally, we send a message to the current user (in the parameters form inquiry to click the OK button is also possible by the combination of keys Ctrl + Enter):



Developer tools

This chapter describes developer tools of the client application Ultimate AEGIS®.

All the basic tools of application developer of the system Ultimate AEGIS® are arranged in the tab "Developer", which is available to any user, which role has *Developer* permission:



The tools are broken down by groups:

- the first group of the tab - tools for working with the system for version control. The system for version control and tools for working with it are described in the chapter Version control;
- *Scripting* – tools for working with commands and scripts. In the chapter Scripts the mechanisms are also described for access to data and special managers;
- *Metadata* – tools for working with metadata objects;
- *Fast access* – tools for fast access to the objects;
- *Apps and modules* – tools for working with applications and modules. In addition to the tools, in the chapter Applications and modules, the peculiarities are described for creation, modules, screen forms as well as specific control elements;
- *Misc* – other tools.

In addition to these tools, the section describes also:

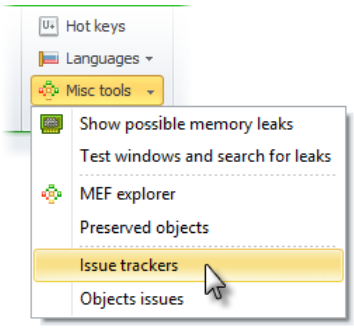
- translation of exceptions;
- predicative access;

- procedures for logging control;
- Tools of tracing, used for control of step-by-step execution of application;
- Description of KERNEL scheme.

Metadata

Metadata group tools are intended for creation and editing metadata objects describing activities of the company:

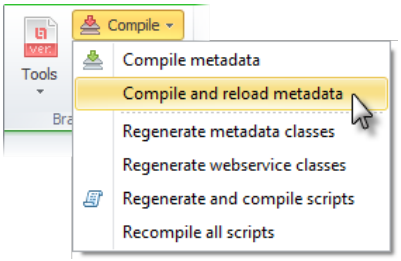
- dictionaries and link tables
- documents and table parts
- totals and etc.



Metadata in particular consist of objects in the DBMS, however DBMS objects aren't directly created by these tools. They only generate script intended for creation of DBMS objects of SQL-script. Respectively, deleting metadata objects from the system won't entail physical deleting objects associated with them from the database (if they were created).

Creation of new metadata object assumes three stages:

1. as a result of creation and saving a new metadata object by means of the Metadata group tool, in the system Ultimate AEGIS® object class by means of which the metadata object will be provided in an application server will be generated;
2. After compilation and reload of meta data the generated class can be used when writing, for example, scripts or handlers. Also object open command will be available;
3. and, at last, metadata objects will be created in DBMS after SQL-script execution.



By means of SQL-script generated by Metadata group tools the following DBMS objects are created:

- tables;
- table view which are used for localization of objects and support of time zones (for *DateTime data type*);
- triggers;
- RLS functions responsible for the access checks and predicative access.

To the discretion of an application developer it is possible not to create excess objects in DBMS, for example, RLS function if metadata object doesn't assume use of predicative access, or table views with appropriate triggers if the metadata object isn't localized and has no *DateTime* properties. However any decision of this sort should be made circumspectly.

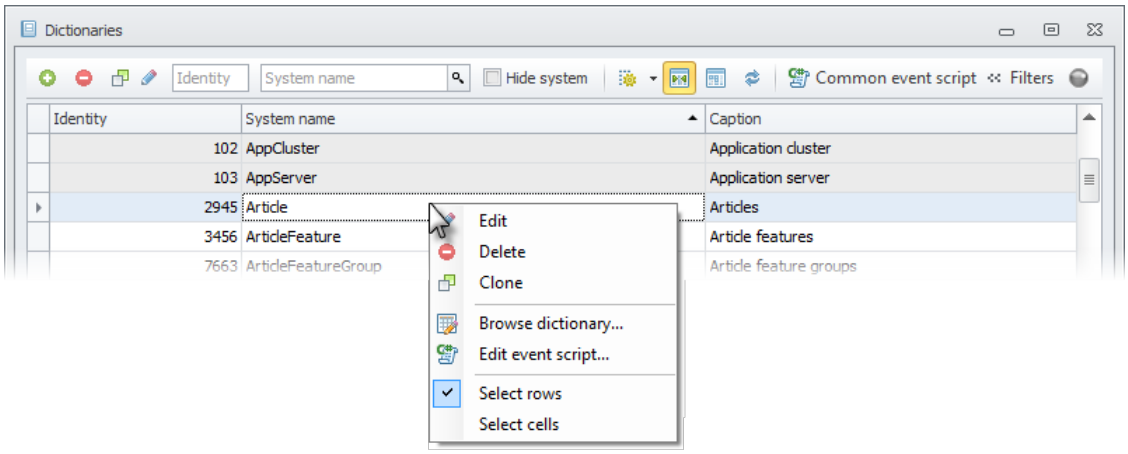
SQLscript needs to be executed in the *Ultima scheme* in any application for work with the relational databases supporting SQL, for example, PL SQL Developer and TOAD.

Just before the script execution it is necessary to specify on behalf of what user Ultimate AEGIS® and on what branch of meta data changes are made. For this purpose it is necessary to execute KERNEL.SET_LOGIN method having specified the user login and an application server code were given:

```
EXEC KERNEL.SET_LOGIN('UserLogin', 1)
```

Dictionaries

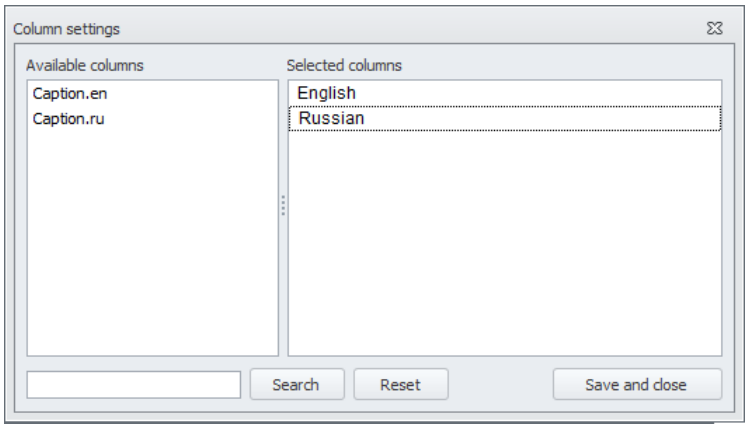
You can view existing and create new dictionaries in the *Dictionaries*:



System (kernel) dictionaries are highlighted with gray color. The application developer has no permissions to edit or delete them, but has a permission to read.

The dictionary records can be filtered by Dictionary *name (System name*). The system dictionaries can be hidden, having set a flag *Hide system*.

In selection of analytical columns it is possible to select localized columns of the name (*Caption*) in all languages that are available from system:

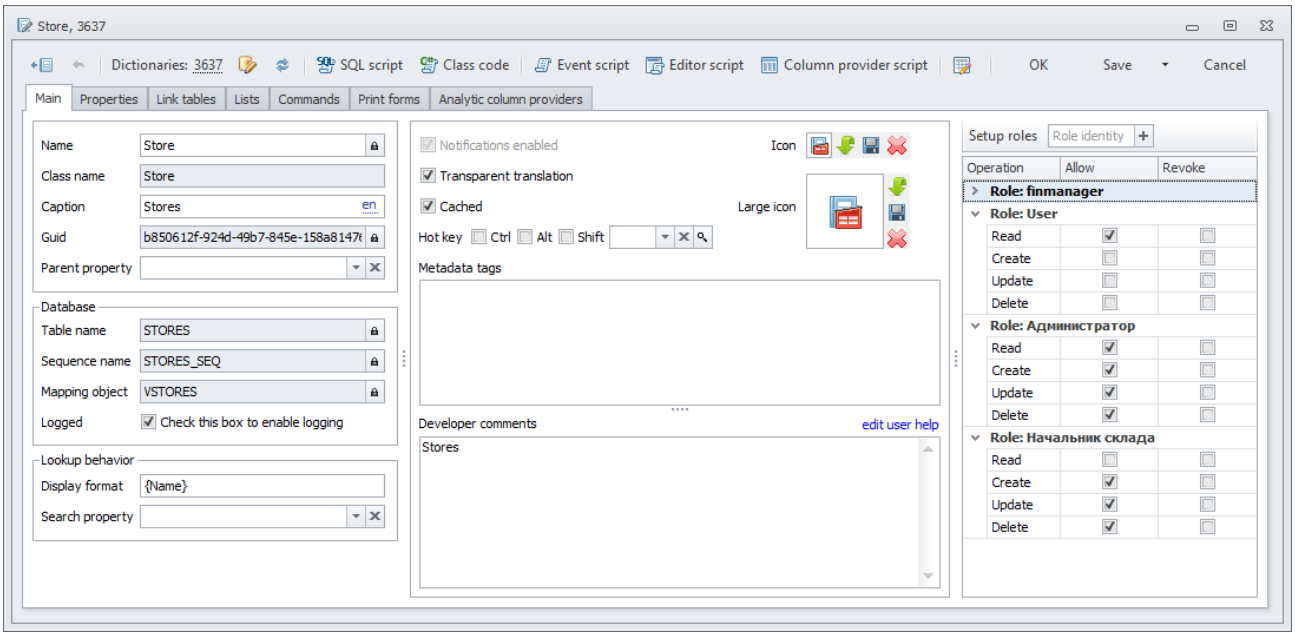


by clicking Common event handler in a toolbar the script of a dictionary event handler is available.

Arrangement of storage for the structure of dictionaries at the database level is detailed in the section Dictionaries in the chapter KERNEL scheme.

Cloning of metadata objects such as dictionaries and link tables is detailed in the Metadata cloning section.

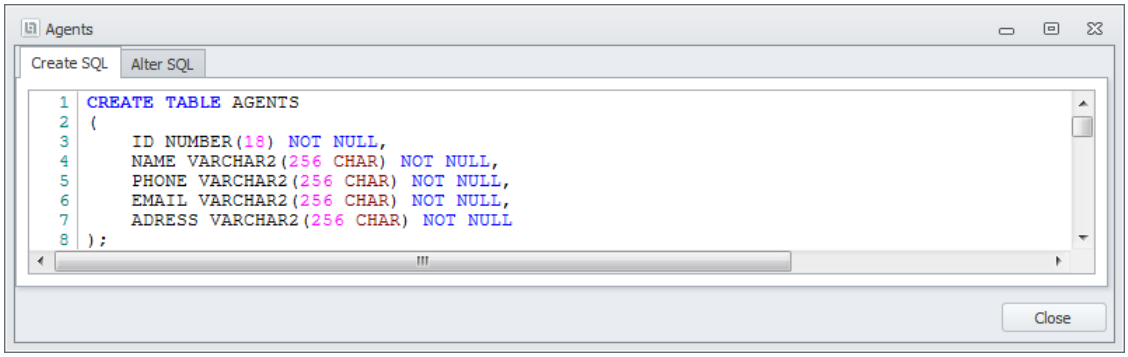
For convenience dictionary properties are grouped in tabs in the form of its structure editing:



The dictionary class name and its ID are displayed in the form heading.

In the form toolbar in addition to the dictionary identifier (it is appropriated automatically) there are buttons:

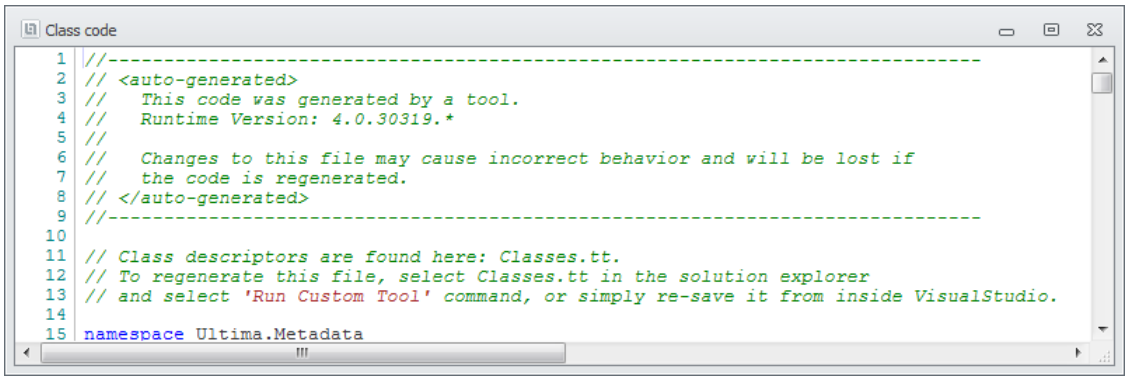
SQL script — SQL script is intended to enter dictionary objects in the database:



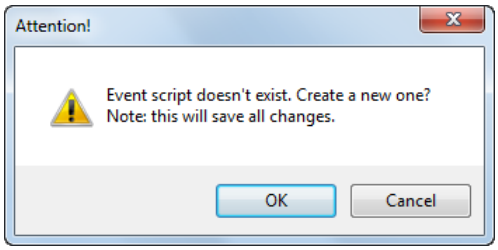
- in the Create SQL tab there is a script used to enter dictionary objects in the database. This script can be applied if the object is created in the DBMS for the first time;;
- in the Alter SQL tab there is a script used to change earlier created dictionary objects in the database.
- buttons on the control bar located in the upper part of the form allow:
  - — to execute the current script in the database,
  - — to copy the current script in a clipboard,
  - — to save the current script in the file on a disk.

Before script execution it is necessary to add rights for the created dictionary in an user role. If there are no dictionary rights, some operations in the database can be unavailable.

Class code — class that describes a dictionary record will be generated in C#:



Event script — a script dictionary event handler. During creation of new dictionary, the events handler is not created. The system suggests to create it by first click. At the same time all changes entered to the object will be saved:



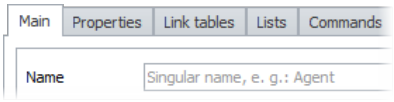
The script of the dictionary event handler can also be opened through the shortcut menu of the dictionary list form.

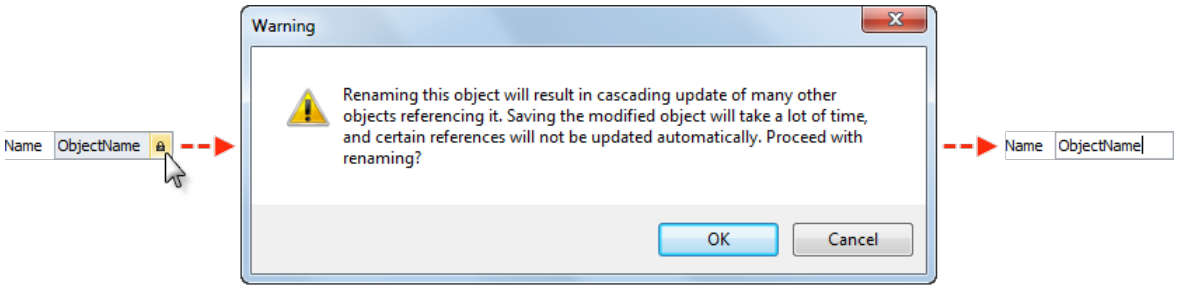
Column provider script — script of column provider of a dictionary. During creation of a new object, the column provider is not created. The system suggests to create it by first click. At the same time all changes entered to the object will be saved:

— a button of opening of a list form of the edited dictionary (functionality works only for already created dictionaries). Also it can be opened through the shortcut menu (item *Browse dictionary...*) of the dictionary list form.

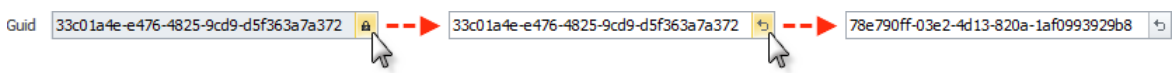
In the Main tab main dictionary properties are organized:

- *Name* – dictionary name. It shall correspond to one dictionary record name (to be in singular). In an example it is a dictionary name *Store* — *Store*. If necessary, it can be changed:





- *Class name* — dictionary class name. It is generated automatically based on Dictionary *Name*;
- *Caption* — that is displayed in screen forms (for example, in title of the list form) is a dictionary name. It is generated automatically based on Dictionary *name* by its partition on single words and transfer to plural. For example for a dictionary with the name *AgentAddress* this property will have meaning *Agent addresses*;
- *Guid* — is used to identify a menu item.
  Guid is generated automatically at random and, if necessary, (in case of coincidence with Guid of another object) can be changed:



- *Parent property* — allows organizing the dictionary in a tree structure (not a mandatory option) if it is specified a property link of the dictionary referring to it as value of this property. It is recommended as the name (*Name*) of this property link to use *ParentID* value. Besides this property shall be not mandatory (a *Not null* flag shan't be set for it);
- property group *Database* — dictionary object name *in the application* database scheme:
  - *Table name* — dictionary table name.
  - *Sequence name* — Sequence name used for identifier (ID) generation of dictionary records;
  - *Mapping object* — the object name on which LINQ requests are displayed. By default this is name of the generated view.
  - *Logged* — determines if the logging of this dictionary is enabled. When this flag is checked, the generated dictionary script includes the calls to the PACK_LOG package enabling the logging of the chosen dictionary properties. The default value is true (checked).

  Property group names *Database* are generated automatically based on *Dictionary name* and if necessary they can be changed:

  Names may contain only letters of the Latin alphabet, digits and a sign "_". At the same time the name shall begin with a letter and its length shan't exceed 30 characters (available quantity of characters is displayed in a control element).

  It is possible to replace view with the table to increase productivity, however in this case multilingualism and translation of time won't be supported. If any of these functions aren't required, and the table is very strongly loaded, it is possible to refuse from views;
- Property group *Lookup behavior*:
  - *Display format* — a format in which dictionary records are displayed in the screen forms, when they are produced not in table but in the row form (for example, in control elements).

    To receive at the output a string as "13, name" where the first value are ID records, and the second separated by a comma — value of Name property, specify a string "{ID}, {Name}" as field value *Display format*.

    Date formatting is supported, for example, for field value *Display format* "{DateTime:d}"{DateTime:d}" only date without time will be removed (information on all available formats of dates can be found on the website MSDN eng/rus);
  - *Search property* — dictionary property among which values search is carried out in a dictionary list form or in a control element;
- *Notification enabled* — a flag allowing to send notifications among all cluster servers on entering changes into dictionary data. For cashed dictionary, the notifications are distributed always irrespective of flag status; If the dictionary is often changeable, it can lead to essential growth of traffic between servers and clients;
- *Transparent translation* — a flag of localization transparency (with the set flag the dictionary is transparent localized);
- *Cached* — a caching flag. With the set flag the dictionary is cached on computers of end users. In case of its repeated retrieval, the data are taken from the local copy but not from database server, which can be used for the dictionaries with rarely changed data. For cached dictionary, the notifications of the change are always distributed, therefore dictionary with frequently changed data should not be made cached.
- *Icon* – standard icon (with the size of 16 x 16 pixels).

  Icons are displayed, for example, in the main menu or in the window title of a list form and a dictionary edit form.

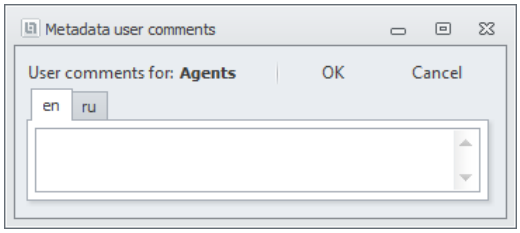
  The buttons to the right of icon preview area allow:

  – loading the icon;

  – saving the icon previously downloaded to the computer;

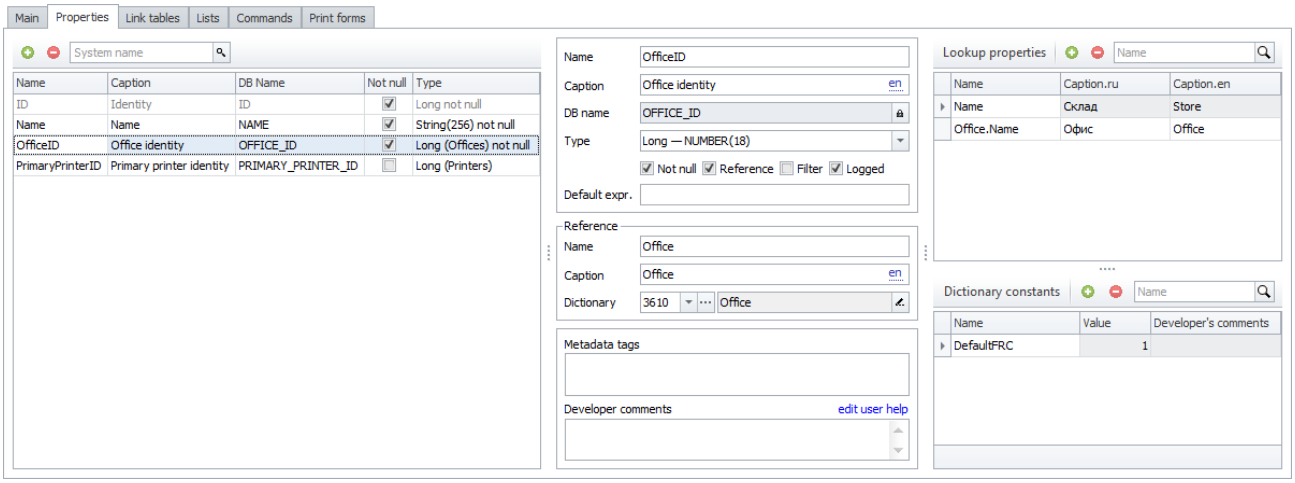
  – deleting the icon;

- *Large icon* – a large icon (with the size of 32 x 32 pixels);
- *Hot Key* – shortcut keys, which call the opening command of a dictionary list form. Using the flags, one or several functional keys (Ctrl, Alt and Shift) can be selected, and a symbol key can be selected in the control element to the right of them.
  The buttons of the control elements, using which symbol selection is made, allow:
  – to select a symbol;
  – to delete the selected value;
  – to view, if such shortcut keys occur for any other command:
- *Metadata tags* – tags used to describe dictionary functionality.
  For example, in the next configuration version some delivery functionality *(a delivery* tag) was, which includes several new dictionaries, document types, changes in a row of system existing objects, etc. If the dictionary is marked with this tag, further it can be found by mean of this tag in the list of the other objects realized under this functionality.
  The tag is added by keys Space or Enter. Delete – by button after the tag. As the gap is used for the tag input it is possible to replace it with characters "_" or "-" in tags with the name of several words;
- *Developer's comments* – comments of the application developer;
- *edit user help* – comment to the object (in this case to the dictionary) which the end user can see in the form of a (hint) which drops down after mouseover. The comment is entered for any language of the system;



- *Setup roles* — a role list for fast draft right set up for the dictionary. Opposite to each operation there are Allow and Revoke flags, which respectively include or turn off access to the dictionary. The role list allows checking easily whether the rights for the dictionary are given at least somebody in system.

In the Properties tab a dictionary property list is at the left, in the middle — all parameters of the property selected at the left, at the right — a list of dictionary properties *Lookup properties*, which will be displayed in the dropdown list of control elements (for example, DictionaryLookupEdit or DictionaryMultiSelectEdit):



The properties can be added or removed using corresponding buttons in the toolbar of the tab: Also properties can be filtered by *Name* (*Name*).

Each property has:

- *Name* – name;
- *Caption* – a name displayed in the screen forms; It is generated automatically based on property *Name* by its partition on single words;

- *DB name* — name of the appropriate field of the dictionary table in the *application database scheme*. Name is generated automatically based on Property *Name;* If it is necessary change it in the same way, as well as names of dictionary objects in the database. Field name has the same restrictions, as other DB objects names have;
- *Type* – property type (see details in section Data types). Depending on type a row of specific parameters can be available:
  - Property *Reference group* (it is available to data type *long,* marked by *Reference* flag):
    - *Name* – a reference name;
    - *Caption* — reference description. It is generated automatically based on reference *Name* by its partition on single words;
    - *Type* — reference type: Dictionary or document);
    - *Dictionary/Document* — a dictionary or a document type which property refers to. When a document type specifying, a typed reference to the document will be created. When record browser opening a relevant document log will be opened, in case of column selection it will be possible to add a column from a title of the appropriate document type;
  - *Max Size* (available for the types of data *Text* and *String*) – limits the size of property value by the specified value;
  - *Multilanguage* (it is available to data types *LargeText*, *Text* and *String*) — a flag specifies that property is Multilanguage;
- *Not null* – a flag indicating if the property is mandatory for fill-in;
- *Reference* (it is available to data type *long*) — a flag specifies that property is a link to dictionary or a document;
- *Filter* — specifies that this property will be displayed by default in the filter of the dictionary list form;
- *Logged* — specifies if logging of this property should be enabled (if logging is enabled for the given dictionary);
- *Default expr.* — C# expression creating property value by default. Expression value is added in property automatically during creation of a new dictionary element.
  For example, it is possible to output an information message as value *Text* property:

```
“Enter as this field value current date in the YYYY-MM-DD format where YYYY is a year, MM - month, and DD - day"
```

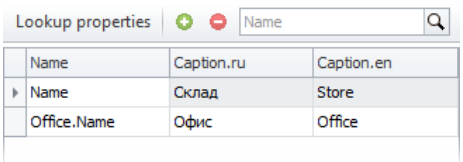
  Or it is possible as value of the same property directly to remove current date having specified the following expression as *Default expr.* parameter value:

```
DateTime.Now.ToString("yyyy-MM-dd HH-mm-ss")
```

- *Metadata tags* – tags used to describe dictionary property functionality. They are similar in meaning to dictionary tags. Used for searching the objects implemented for certain functionality associated with such tag.
- *Developer's comments* – comments of the application developer;
- *edit user help* – comment to the object (in this case to the dictionary property) which the end user can see in the form of a hint which drops down after mouseover. The comment is entered for any language of the system;

In the *Lookup properties* list there are properties which will be displayed in the dropdown list of control elements (for example DictionaryLookupEdit or DictionaryMultiSelectEdit). If this list is empty, properties listed in *Display format* (on “Main” tab) will be displayed. *Lookup* properties can be added or deleted by appropriate buttons in a toolbar, and also filtered according to *Name* (*Name*).

In the *Lookup properties* list both dictionary properties and property of other dictionaries to which this dictionary refers by means of properties links can be listed. In an example above mentioned — this is *Name property of the Store dictionary* and *Name property of the Office* dictionary specified in the *Office.Name* format.



Property name are specified in the list column *Name* and in the columns *Caption* their localized names for display in screen forms (control elements) for each of system languages are specified.

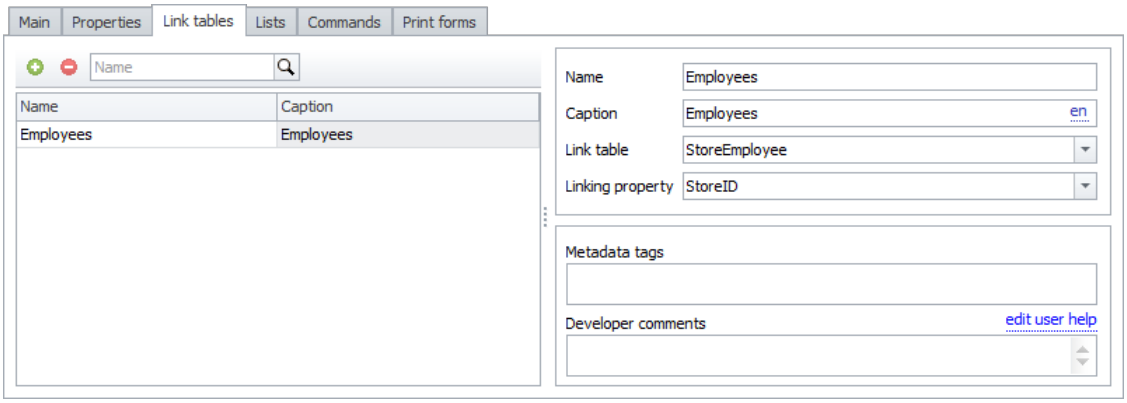
In the *Dictionary constants* list dictionary constants are listed. Constantscan be added or deleted by appropriate buttons in a toolbar, and also filtered according to *Name* (*Name*).

Dictionary constant existence removes the application developer the need to hard code record codes or to create for this purpose separate constants in the constant dictionary:

```
record.TypeID = DictionaryName.Constants.ConstantName;
```

Names and values of constants are specified in the list columns *Name* and *Value* and in the column *Developer's comments* — comment of an application developer.

In the "Link tables" tab there is a list of link dictionary tables at the left, at the right there are all parameters of the table selected at the left:

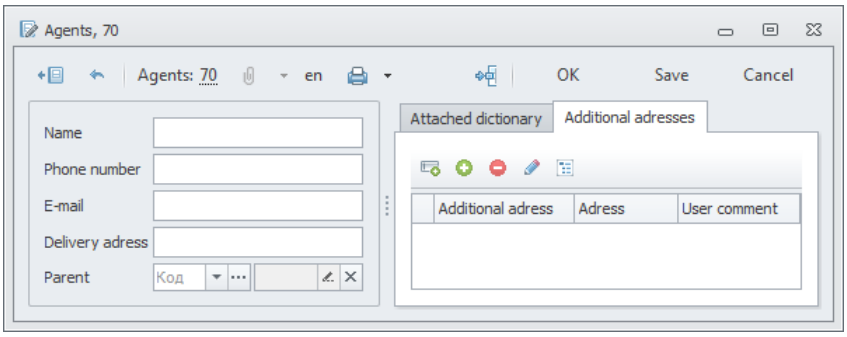


Link tables can be added or removed using corresponding buttons in the toolbar of the tab: Also link tables can be filtered by the *Name* (*Name*).

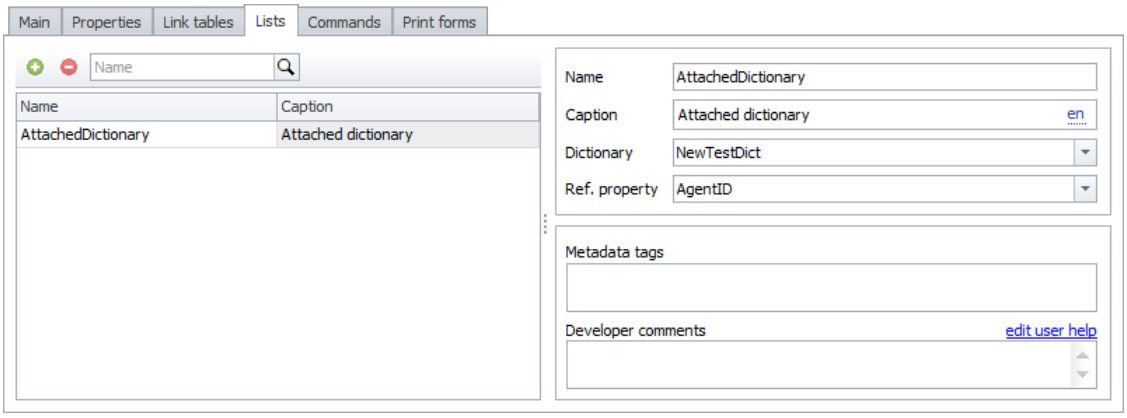
Each link table has:

- *Name* – name;
- *Caption* – a name displayed in the screen forms; It is generated automatically based on link table *Name* by its partition on single words;
- *Link table* — directly a link table;
- *Linking property* — a property of the link table which refers to this dictionary;
- *Metadata tags* – tags used to describe link table functionality.
- *Developer comments* – comments of the application developer;
- *edit user help* – comment to the object which the end user can see in the form of a hint which drops down after mouseover.

Before adding a link table to the dictionary, it needs to be created (see Link tables). It is possible to select a link table only among those which property link refers to the current dictionary. If adding the link table to the dictionary, the user in the dictionary edit form will be able on a separate tab with the name corresponding to *Caption* property to see the list of the link table records associated with it, to add new records in it, to edit and delete existing records:



In the "Lists" tab the list of embedded dictionaries is located, on the right all parameters of the embedded dictionary selected at the left:

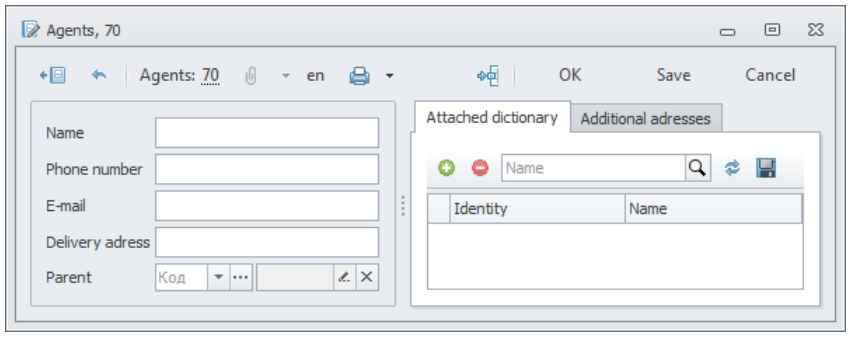


Embedded dictionaries can be added or removed using corresponding buttons in the toolbar of the tab: Also embedded dictionaries can be filtered by the *Name* (*Name*).

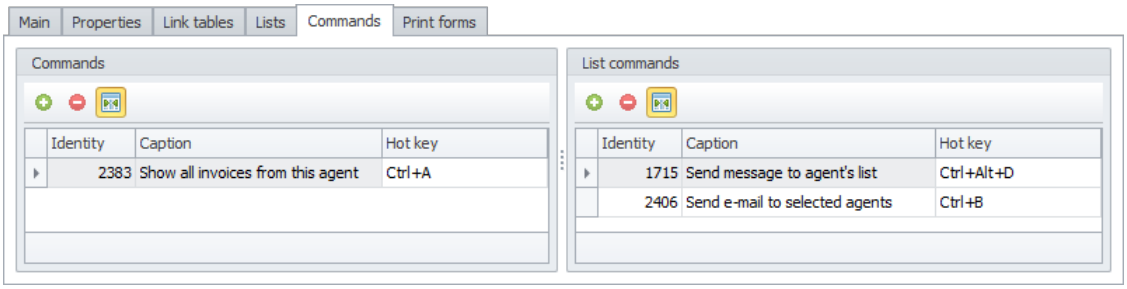
Each embedded dictionary has:

- *Name* – name;
- *Caption* – a name displayed in the screen forms; It is generated automatically based on embedded dictionary *Name* by its partition on single words;
- *Dictionary* — directly the embedded dictionary;
- *Ref. property* — a property of the embedded dictionary which refers to this dictionary;
- *Metadata tags* – tags used to describe the embedded dictionary functionality.
- *Developer's comments* – comments of the application developer;
- *edit user help* – comment to the object which the end user can see in the form of a hint which drops down after mouseover.

Before adding a embedded dictionary, it needs to be created. It is possible to select a embedded dictionary only among those which property link refers to the current dictionary. When the embedded dictionary adding the user in the dictionary edit form will be able on a separate tab with the name corresponding to *Caption* property to see the list of the link table records associated with it, to add new records in it, to edit and delete existing records:



In the "Commands" tab there is a command list, available to this dictionary:

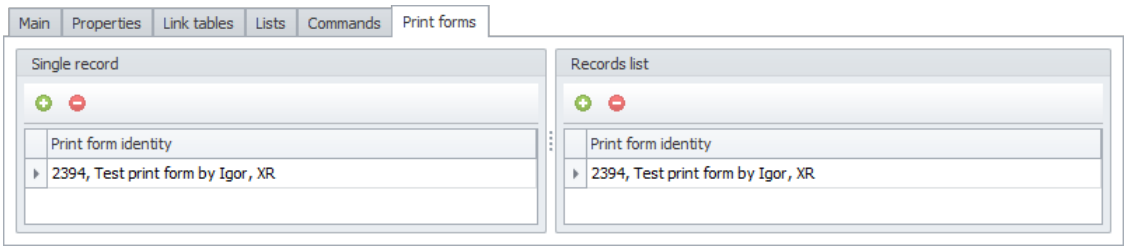


In the left part of the tab the "Commands" list over one dictionary record (which are available through the shortcut menu in the dictionary list form or from the record edit form) is located, in the right part — the List commands over several dictionary records (marked in the dictionary list form) is located.

The commands can be added or removed using corresponding buttons in the toolbar: When adding the command edit form will open. When deleting the command will be deleted not only from the list, but also from the appropriate command dictionary.

Commands added through list forms of the appropriate dictionaries will be automatically displayed in these lists.

In the "Print forms" tab there is a list of printing forms used for this dictionary:



In the left part of the tab the "Single record" list of print forms used when one dictionary record printing (they are available when printing from the form of dictionary record edit) is located, in the right part — the "Records list" — the print forms used when printing several dictionary records printing (marked in its list form).

Print forms can be added or removed using corresponding buttons in the toolbar of the tab: When adding the list form of print forms of the dictionary will open wherein it will be possible to select by double left-click a necessary print form. When deleting in the appropriate printed form for the the dictionary *Assign* flag will be deselected (a print form at the same time won't be deleted).

Print forms assigned to the dictionary from the edit form of print forms will be automatically displayed in these lists.

Dictionary record class

During creation of each dictionary, a dictionary record class is generated. Its initial description is represented with dictionary *Name*, and the list of its properties is represented with corresponding dictionary properties.

For instance, let us consider creation of simple dictionary *DictionaryName* with the properties *ID*, *Name* and *ReferenceID*.

The model class of the subject area, generated according to this description, looks like as follows:

```
public partial class DictionaryName : IDictionaryRecord
{
	public long ID { get; set; }
	public string Name { get; set; }
	public long ReferenceID { get; set; }
}
```

All classes of dictionary records (and only they) implement *IDictionaryRecord* interface. Therefore, a list of all classes of dictionary records can be obtained by requesting who implements this interface:

```
public interface IDictionaryRecord : IEntity, IBusinessObject
{
	// Returns the link tables associated with the dictionary record.
	IKeyValueStore<string, ILinkTable> LinkTables { get; }

	// Returns the collections of dictionary records associated with the dictionary
record.
	IKeyValueStore<string, IDictionaryTable> DictionaryLists { get; }
}
```

The field of type *EditableValue<T>* corresponds to each dictionary property, where T is one of types indicated in metadata:

```
private EditableValue<string>name; ///field

public string Name ///property
{
	get { return name.Value; }
	set { name.Value = value; }
}
```

A collection of type *DictionaryTable<T>* may also correspond to the property (where *T* is a type of collection element).

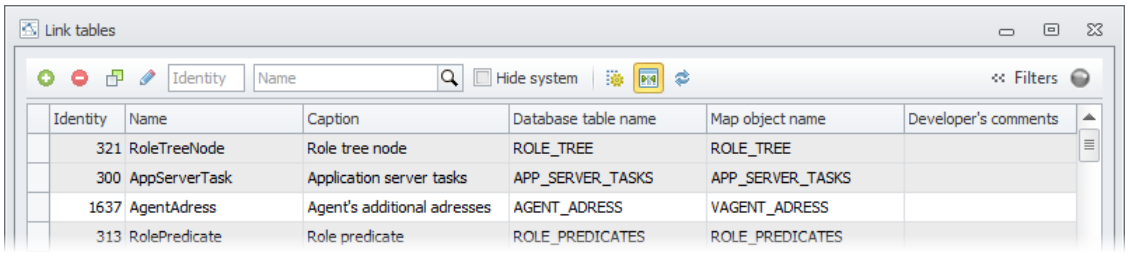
Example of use:

```
[Import]
private IDictionaryManager DictionaryManager { get; set; }

// Get the dictionary record.
var dictionaryRecord = DictionaryManager<DictionaryName>.GetRecord(10);

// Get the value of the property Name of dictionary record.
var name = dictionaryRecord.Name;

// Get the link table of dictionary record.
var linkTable = dictionaryRecord.LinkTableName;
```

Link tables

Link tables are used for keeping links between the dictionaries in the system.

Viewing existing and creating new link tables can be made in the dictionary Link Tables:



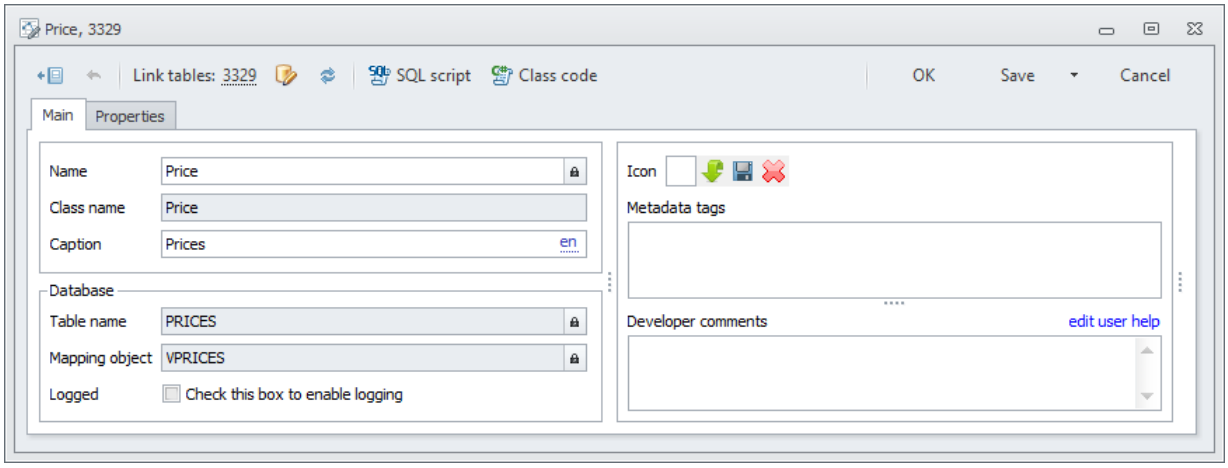
The system (kernel) link tables are highlighted with grey in the list. The application developer has no permissions to edit or delete them, but has a permission to read.

The dictionary records can be filtered by *Name* of the link table. The system link tables can be hidden, having set the *Hide system* flag.

Arrangement of storage for the structure of link tables at the database level is detailed in the section Dictionaries in the chapter KERNEL scheme.

Cloning of metadata objects such as dictionaries and link tables is detailed in the Metadata cloning section.

The form for editing of the structure of link tables has lower number of properties as compared to the form for editing dictionaries:

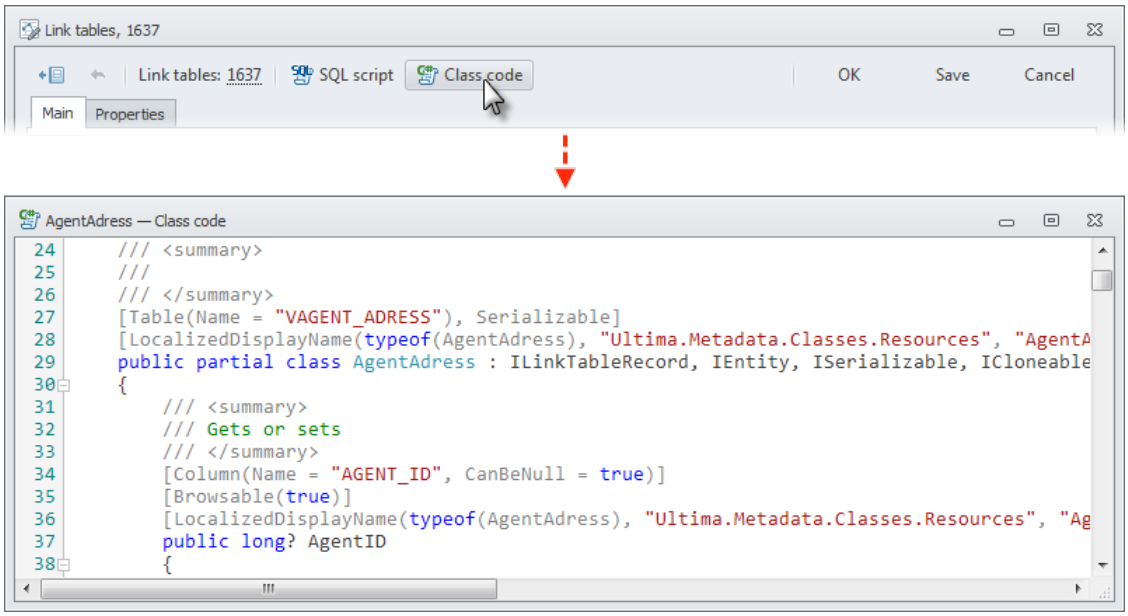


The link table class name and its ID are displayed in the form heading.

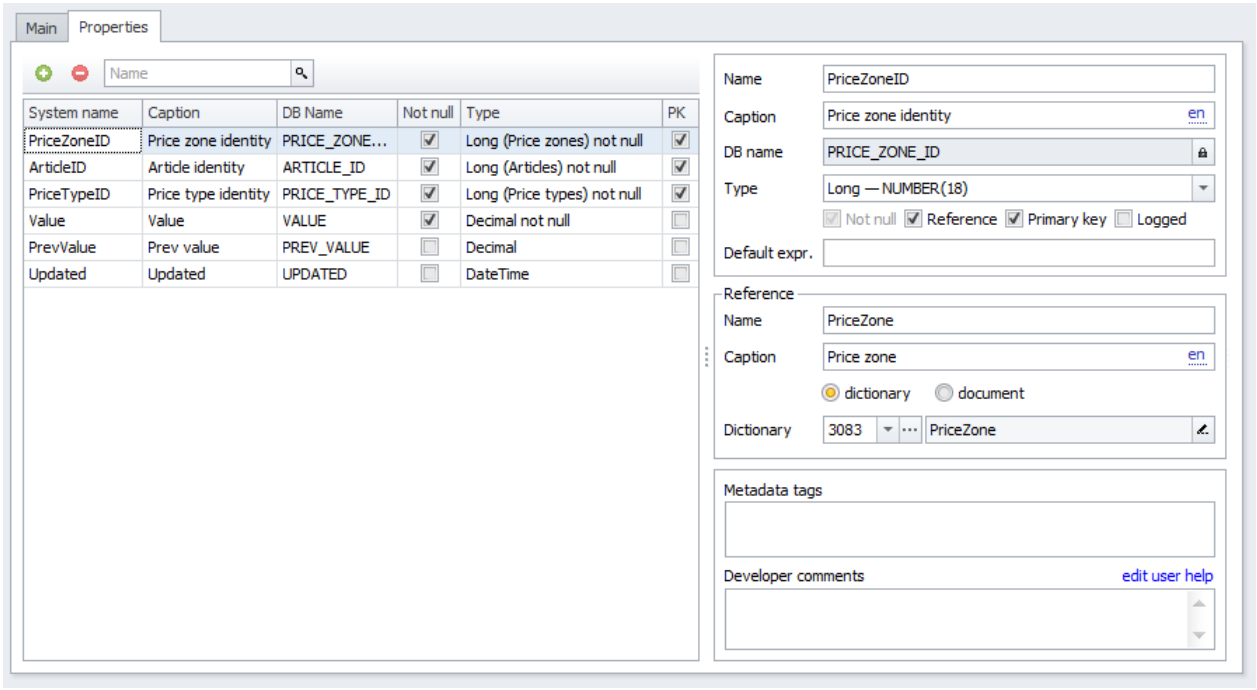
As distinguished from the dictionary, the link tables do not support multilanguage option, have no list form and cannot be added to the main menu. As a result, they do not require hot keys, guid and large icon.

Using SQL script button, you can view SQL script for creation or change of the objects of DBMS table part.

Using Class code button, you can view the class generated in C#, describing the link table record:



The properties of the link table are described on the whole similarly to dictionary properties. The only difference is in the *Primary key* flag, which indicates that this field is included into the primary key (for a dictionary, the primary key is always represented with the *ID* field). For such fields, the *Not null* flag is automatically checked as well, and it cannot be unchecked manually:



To link three dictionaries by using a link table (the above screenshot represents *PriceZonesID, ArticleID* and *PriceTypeID*), the three properties of the link table should be made as links to these dictionaries. For the property-link in the brackets after the type (the link is always of *long* type), the dictionary name, which it refers to, is indicated.

Now to provide the end user with a possibility to fill in the data of the link table, in the tab Link Tables, the created link table should be added to the record of corresponding dictionary. If the users should be provided with a possibility to fill in the link table with the data from any of the dictionaries, it should be added to both, if only from one – into one.

Thus, three and more dictionaries can be linked using the link table.

In addition to the properties-links, creation of common properties can be also made in the link table. In this case, the user will be able not only to link the dictionary records between themselves but associate that link with the data set.

Arrangement of the products price lists can be furnished as an example of such link. Two dictionaries – *Products* and *Price columns* – are interconnected between themselves using a link table *price list* , which in addition to two properties-links to these dictionaries contain additional property *Price*. The user supports it with price input while selecting a column for particular product.

If the product must have the only price in each price column, the properties-links of the link table *price list* should be made key ones having set *Primary Key* flag for them. In this case, it will be possible to add only one record to the link table for each *Product-Price column* set.

Class of link table record

During creation of each link table, a class of link table record is generated. Its initial description is represented with link table *Name*, and the list of its properties is represented with corresponding link table properties.

For instance, let us consider creation of simple link table *LinkTableName* with *ReferenceID*, *AnotherReferenceID* and *Value* properties.

The model class of the subject area, generated according to this description, looks like as follows:

```
public partial class LinkTableName : ILinkTableRecord
{
       public long ReferenceID { get; set; }
       public long AnotherReferenceID { get; set; }
       public decimal Value { get; set; }
}
```

All classes of records of the link tables implement *ILinkTableRecord* interface. Therefore, a list of all classes of records of the link tables can be obtained by requesting who implements this interface:

```
public interface ILinkTableRecord : IEntity
{
}
```

The field of type *EditableValue<T>* corresponds to each property of link table record, where T is one of types indicated in metadata:

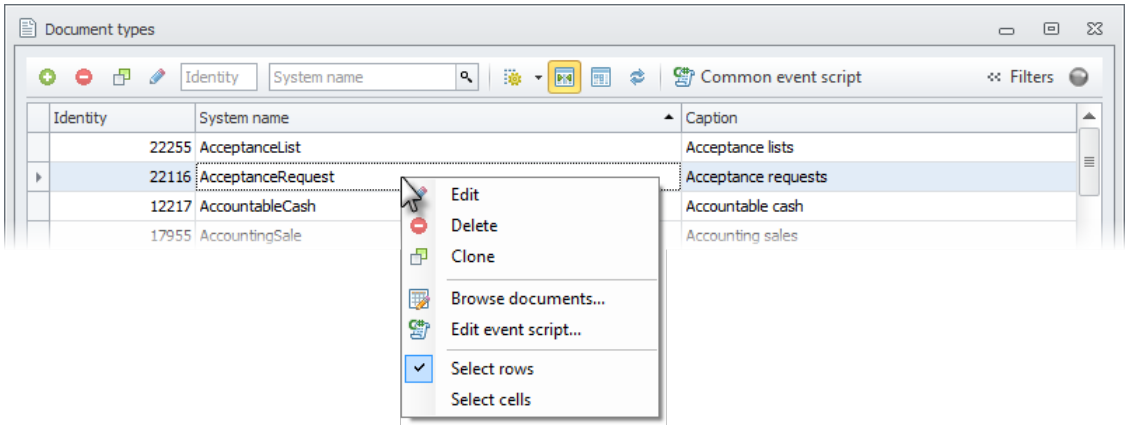
Example of use:

```
[Import]
private ITableSourse DataContext { get; set; }
```

```
var reference = 10;
var anotherReference = 11;

var query =
       from linktable in DataContext.GetTable<LinkTableName>()
       where
              linktable.ReferenceID == reference &&
              linktable.AnotherReferenceID == anotherReference
       select linktable.Value;

return query.Single();
```

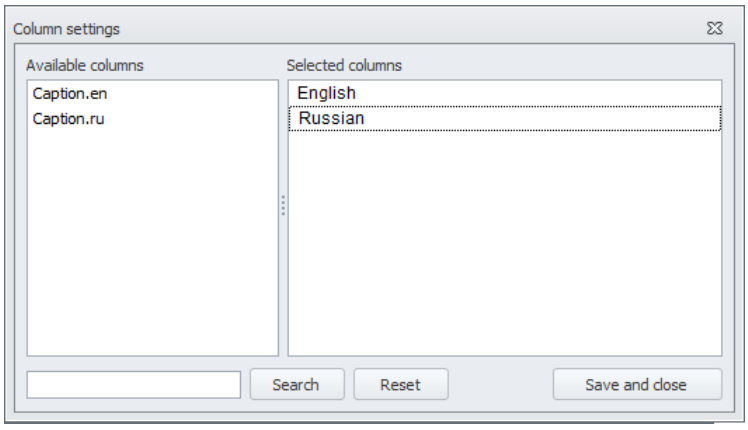
Document types

Viewing existing and creating new document types can be made in the Document types dictionary:



The dictionary records can be filtered by *Name* of document type (*System name*).

The analytical columns selection allows selecting columns captions (*Caption*) localized to all system languages:



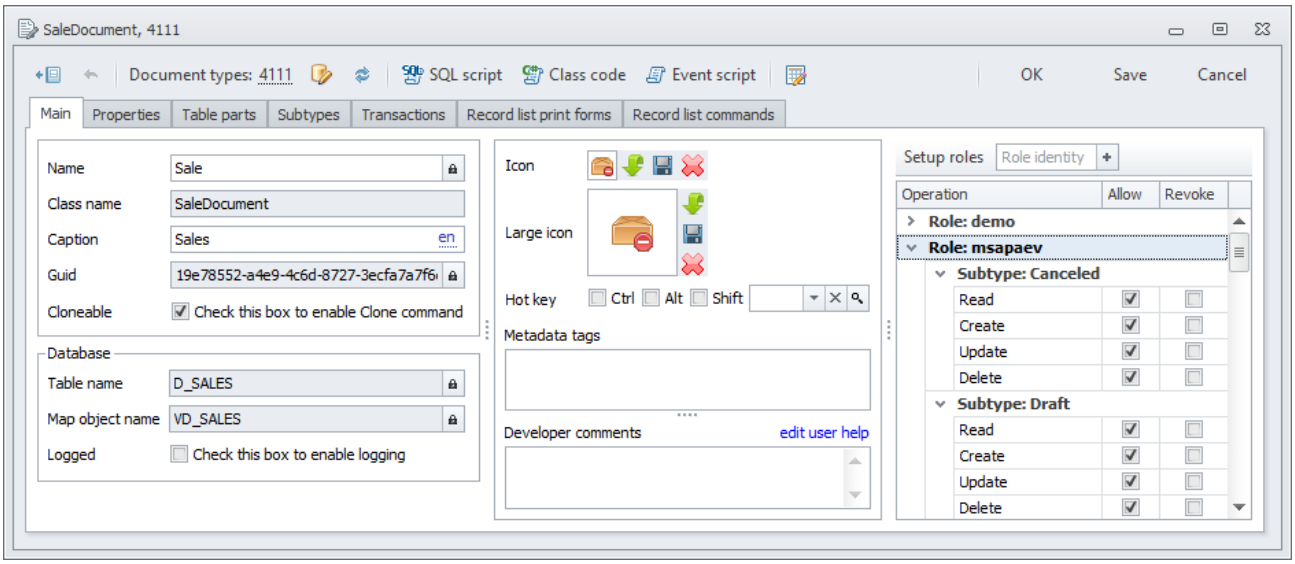
In "Caption" column names are displayed in current user language.

Button Common event handler in the toolbar shows a script of the event handler of all documents.

Arrangement of storage for the structure of documents at the database level is detailed in the section Documents, chapter KERNEL Scheme.

Cloning of metadata objects such as document and table part types is detailed in the Metadata cloning section.

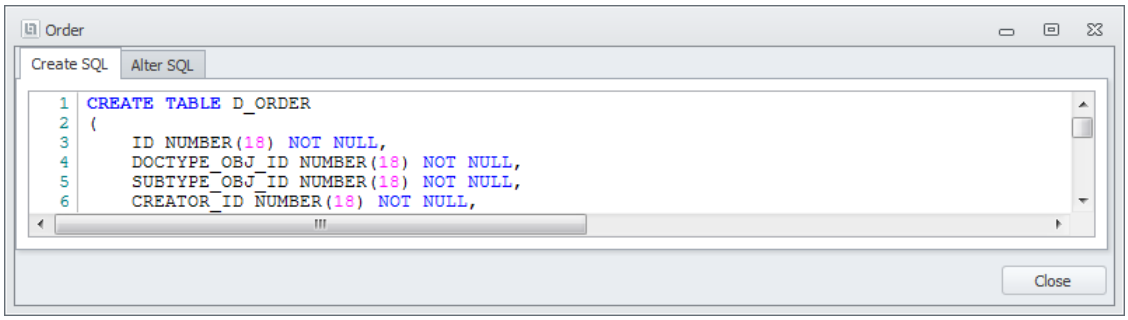
For ease of use, properties of a document type in the document structure edit form are grouped in tabs:



The class name of the document type and its ID are displayed in the form heading.

In addition to the document type ID (assigned automatically), the toolbar includes the following buttons:

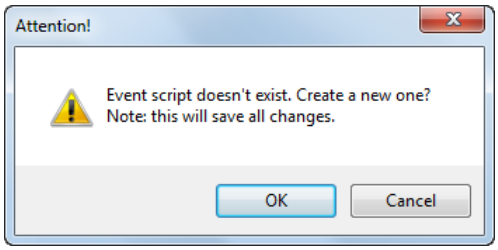
SQL script — an SQL script designed to create document type objects in the database:



- "Create SQL" tab contains a script used for creating document type objects in the database. The script can be used if the object is created in the database for the first time;
- "Alter SQL" contains a script used for changing document type objects previously created in the database;
- the toolbar buttons in the upper part of the form allow:
  - — execute the current script in the database,
  - — copy the current script to the clipboard,
  - — save the current script on the computer.

Class code — a class describing the document type will be generated in C#;

Event script — document event handler script. When creating a new document type, an event handler is not created. The system will suggest to create the handler when click the button for the first time. In the process, all changes made to the object will be saved:

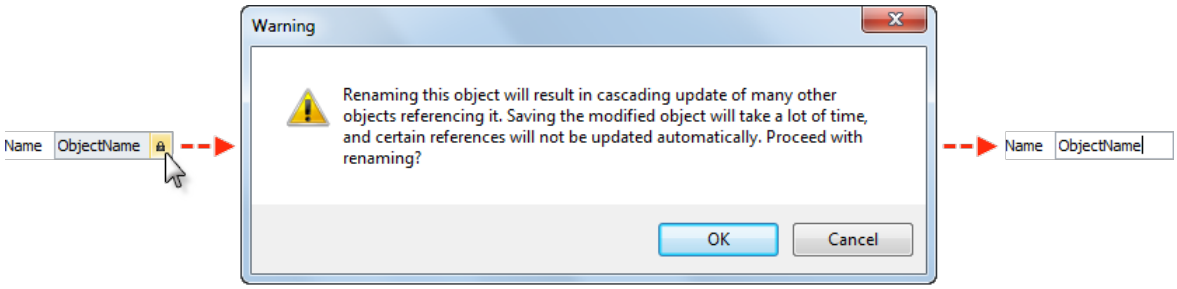


The document event handler script can also be opened via the context menu of the document types list form.

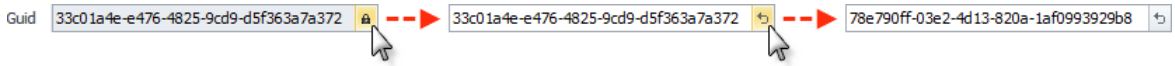
— the button to open the list form of documents of the type being edited (the function is available only for existing document types). The document register can also be opened via the context menu (*Browse documents...* item) of the document types list form.

Main properties of a document type are arranged in the "Main" tab:

- *Name* — document type name. The name must correspond to the name of one document and be in singular). In the example above, this name is *Order*. If necessary, can be changed:



- *Class name* — document type class name. Generated automatically on the basis of the *Name* of the document type;
- *Caption* — document type name displayed in screen forms (e. g., in document registers). Generated automatically on the basis of the *Name* of the document type by dividing it into separate words in plural number. For example, for the *CashPayment* type, the property will be named as *Cash payments*;
- *Guid* — used to identify a menu item.
  Guid is generated automatically at random and, if necessary (in case of coincidence with Guid of another object), can be changed:



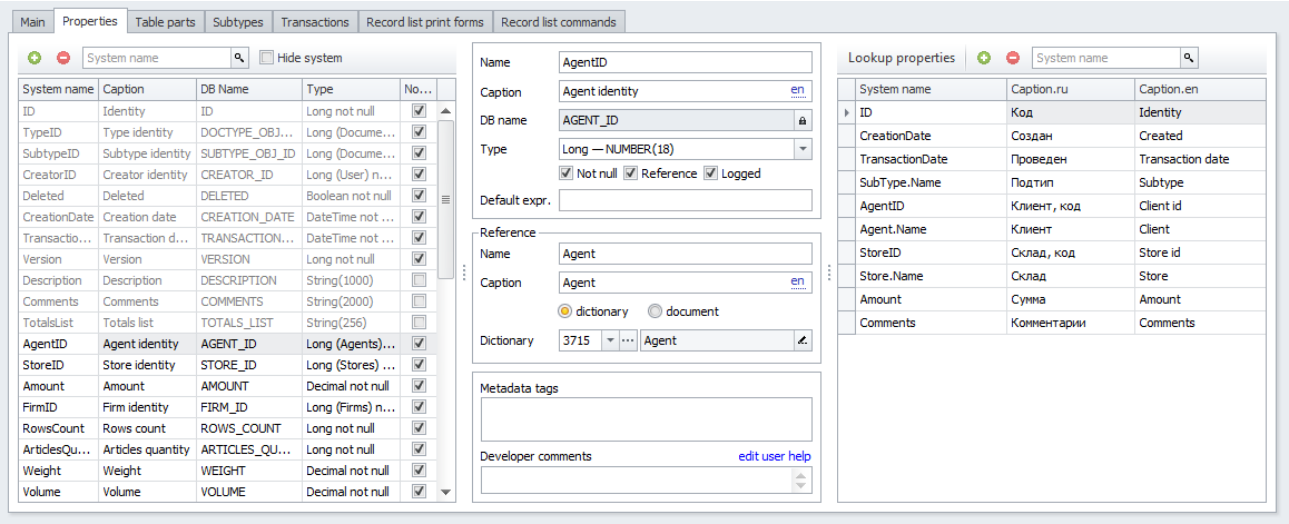
- *Cloneable* — the check box enabling the system command to clone a document (in case of dictionaries, cloning is always enabled; for documents, an explicit permission is required).
- *Database* group of properties — names of document type objects in the database *application scheme*:
  - *Table name* — name of document type table;
  - *Map object name* — name of object for mapping LINQ queries. By default, this is the name of view being generated.
  - *Logged* — determines if the logging of this document type is enabled. When this flag is checked, the generated document script includes the calls to the PACK_LOG package enabling the logging of the chosen document properties. The default value is true (checked).

  Database object names are generated automatically and can be changed, if necessary:

  Name may include Latin letters, numbers and "_" (underskernel character) only. It must begin with a letter and cannot be longer than 30 characters (number of characters left for entering is displayed in the control);
- *Icon* — a standard icon 16x16 pix.
  Icons can be found, e. g., in the main menu or in the window caption of a list form and a dictionary edit form.
  The buttons to the right of icon preview area allow:
  – loading an icon;
  – saving an icon previously downloaded to the computer;
  – deleting an icon;
- *Large icon* — a large icon (32x32 pix);

- *Hot key* — shortcut keys to call a command to open the list form of documents of this type. Using the flags, one or several functional keys (Ctrl, Alt and Shift) can be selected; a symbol key can be selected in the control element to the right of them.
  The buttons of a control element for selecting a symbol allow:
  – select a symbol;
  – delete a value selected;
  – view, if such shortcut keys occur for any other command:
- *Metadata tags* — tags used for description of functions of a document type. Used for searching objects implemented for a particular functional that is associated with this tag.
  A tag can be added by using the keys Space or Enter. To delete a tag, use the button after the tag. Since space is used for entering a tag, it can be replaced by either symbol "_" or "-" in tags consisting of multiple words;
- *Developer comments* — a comment by an application programmer;
- *edit user help* — a comment to an object that the end user can see as a drop-down hint when hovering the pointer over the object. The comment is entered in all system languages;
- *Setup roles* — list of roles for a draft setup of document rights. The rights are arranged in subtypes; each operation is supplied with flags Allow and Revoke, which allow or revoke access to a particular subtype respectively. The list of roles allows checking if there are any rights to work with the document granted to someone.

In "Properties" tab, on the left, there is a list of the document (header) properties; the parameters of a property selected to the left are shown in the center; on the right, there is a list of document properties *Lookup properties*, which are displayed in the drop-down list of the controls:



Properties can be added or deleted by the corresponding buttons in the tab's toolbar. Properties can also be filtered by *Name*. By default, the system properties that are specific to all document types and created automatically will not be displayed (*Hide system* flag).

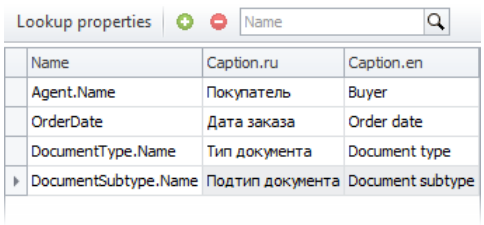
Each property has:

- *Name* — name;
- *Caption* — name displayed in screen forms. Generated automatically by dividing the *Name* into separate words;
- *DB name* — name of a corresponding table field of the document type in the database *application scheme*. The name is generated automatically. If necessary, it can be changed the same way as the names of document type objects in the database. The field name is subject to the same limitations as the names of other database objects;l

- *Type* — property type (for details, see Property types). Depending on a type, a number of specific parameters is available:
  - *Reference* group of properties (available for *long* type of data marked with the *Reference* flag):
    - *Name* — name of reference;
    - *Caption* — caption for reference. Generated automatically by dividing the reference's *Name* into separate words;
    - *Type* — reference type: dictionary or document;
    - *Dictionary/Document* — dictionary or document type that the property points to;
  - *Max size* (available for *Text* and *String* types) limits the size of a property value by the amount specified;
  - *Multilanguage* (available for *LargeText*, *Text* and *String* types) — indicates that the property is multi-language;
- *Not null* — indicates if the property is mandatory;
- *Reference* (available for *long* type) — indicates that the property is a reference to a dictionary or a document;
- *Logged* — specifies if logging of this property should be enabled (if logging is enabled for the given document type);
- *Default expr.* — a C# expression that forms a default property value. The expression value is automatically inserted to the property during the creation of a new document of this type.
  E. g., it is possible to display an information message as a value of the property of the *Text* type:
  ```
  "Enter as a value for this field the current date in the format YYYY-MM-DD, where YYYY is the year, MM is the month, DD is the day"
  ```
  Or you can display the current date at once; to do this, enter as the value of the *Default expr.* parameter the following expression:
  ```
  DateTime.Now.ToString("yyyy-MM-dd HH-mm-ss")
  ```
- *Metadata tags* — tags used to describe the properties functionality. Bear the same meaning as the document type tags;
- *Developer comments* — a comment by an application programmer;
- *edit user help* — a comment to an object that the end user can see as a drop-down hint when hovering the pointer over the object. The comment is entered in all system languages;

In the *Lookup properties* list, there is a list of properties displayed in the drop-down list of the controls. If this list is empty, all document properties will be displayed. *Lookup* properties can be added or deleted with the help of the corresponding button in the toolbar; they can also be filtered by *Name*.

The *Lookup properties* list may include both properties of the document itself and properties of the dictionaries that the given document points to via properties-links. In the example above, such properties are *OrderDate* of the document type *Order*, and *Name* of the dictionaries *Agent*, *DocumentType* and *DocumentSubtype* displayed in the format *Dictionary name.Property name*.



Names of the properties are shown in the list column *Name*; *Caption* columns contain their localized names that are to be displayed in screen forms (controls) for each system language.

In the "Table parts" tab to the left, there is a list of table parts of the document type; on the right, all parameters of the table part selected to the left:

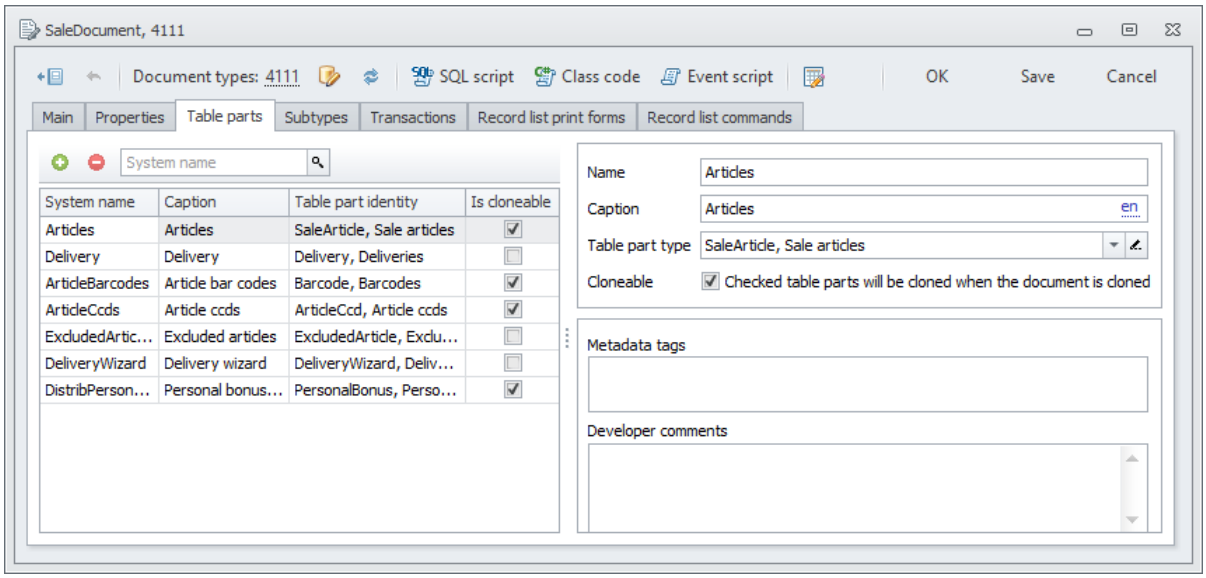
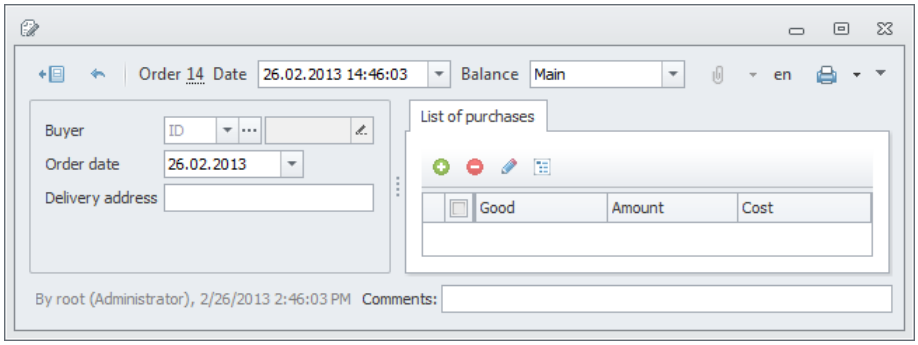


Table parts can be added or deleted with the help of the corresponding buttons in the tab's toolbar. Table parts can also be filtered by *Name*.

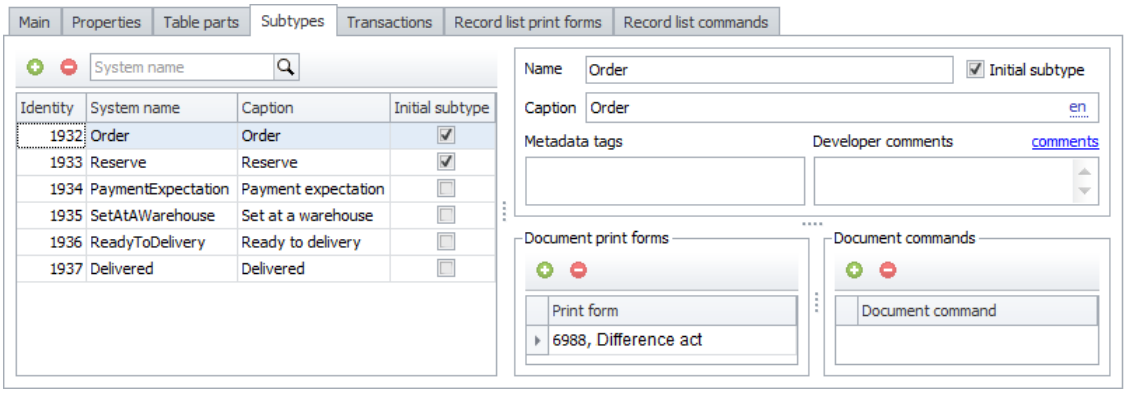
Each table part has:

- *Name* — name;
- *Caption* — name displayed in screen forms. Generated automatically by dividing the *Name* into separate words;
- *Table part identity* — table part per se, the edit form of which is opened by clicking the button in the control;
- *Is cloneable* — indicates that the table part contents will be copied during the cloning. The flag is available only if the cloning of the given document type is allowed by the flag *Cloneable* in the Main tab.
- *Metadata tags* — tags used for description of the table part functionality.
- *Developer comments* — comments by an application programmer.

Before adding a table part to a document type, it must be created (see Table parts). The table part added will be available in the document edit form of this type in a separate tab bearing a name that corresponds with the *Localized name* property:



In the "Subtypes" tab, there is a list of document subtypes; on the right, all parameters of the subtype selected to the left:



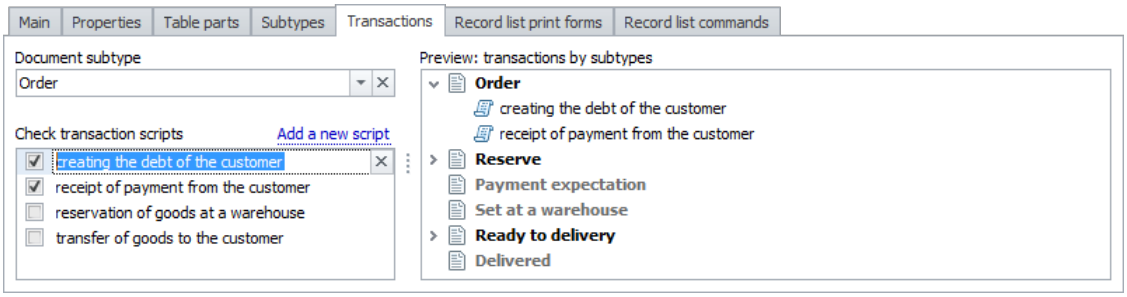
Subtypes can be added or deleted with the help of the corresponding buttons in the tab's toolbar. Subtypes can also be filtered by *Name* (*System name*).

Each subtype has:

- *Name* — system name. When saving a subtype, a constant with the same name is generated for each subtype;
- *Initial subtype* — initial document subtype. When creating a new document, the user will be offered a subtype for the document among the subtypes checked with this flag. If only one initial subtype is specified, it is selected automatically during the creation of a new document.

  There is no way to directly select and change a subtype in the document edit form. Also, one cannot save a document having no subtype selected. Therefore, if no initial subtype is determined for the document type for some reason, it is necessary to select a subtype in the handler before creating the document.
- *Caption* — name displayed in screen forms. Generated automatically by dividing the subtype's *Name* into separate words;
- *comments* — a comment to an object that the end user can see as a drop-down hint when hovering the mouse pointer over the object;
- *Metadata tags* — tags used for description of the document subtype functionality;
- *Developer comments* — comments by an application programmer.
- *Document print forms* — list of print forms used for the given document subtype.
  Print forms can be added or deleted with the help of the corresponding buttons in the control's toolbar. When adding, a list form of print forms will open, where one can select existing print forms or create new ones. When deleting a print form, it will be deleted only from the list of print forms; it will remain in the print forms dictionary. Print forms assigned to a subtype from the print forms edit form will be automatically displayed in this list.
  A print form can be opened in the edit form by double left-click on it in the list;
- *Document commands* — list of commands available for the given document subtype.
  Commands can be added or deleted with the help of the corresponding buttons in the control's toolbar. When adding, a list form of commands will open, where one can select existing commands or create new ones. When deleting a command, it will be deleted only from the list of subtype's commands; it will remain in the commands dictionary. Commands added for a particular subtype via the commands edit form will be automatically displayed in this list.
  A command can be opened in the edit form by double left-click on it in the list.

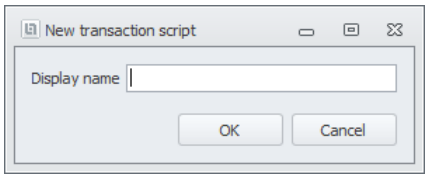
In the "Transactions" tab, a list of transaction scripts of the given document type is located; these scripts from transactions in totals:



For each subtype, it is needed to specify which transaction scripts will be executed while saving the document. Scripts are selected from the common list of scripts tied to the given type.

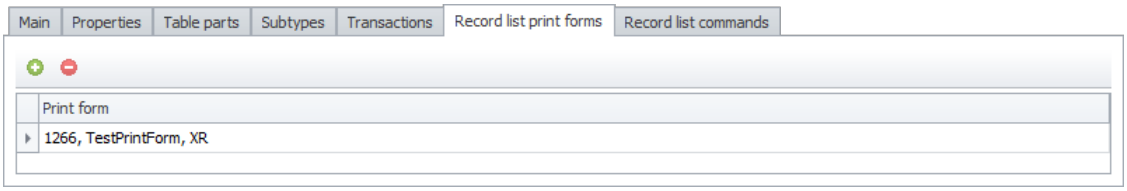
The tab is divided into two parts:

- the left part includes the *Document subtype* drop-down list, at the top of which the document subtypes are specified. At the bottom of the *Check transaction scripts* list, there are all available transaction scripts specified; those checked with flags will be executed while saving the document in a subtype selected in the *Document subtype* list.
  - to add new transaction scripts, click the link *Add a new script*:



    Scripts are created after the document type has been saved; after that, the scripts become available for editing.
  - to rename a script, left-click on its name;
  - to open a script in the edit form, click the button ··· to the right of its name;
  - to delete a script, click the button ✕ to the right of its name. At the same time, the script must not be tied to any document subtype;
- at the tab's right part (*Preview*), one can see a tree structure for all document subtypes and transaction scripts associated with them. Names of subtypes with no transaction script tied to are shown in **gray**. Click left mouse button on the name of a newly-created script in Preview area will open the script edit form.

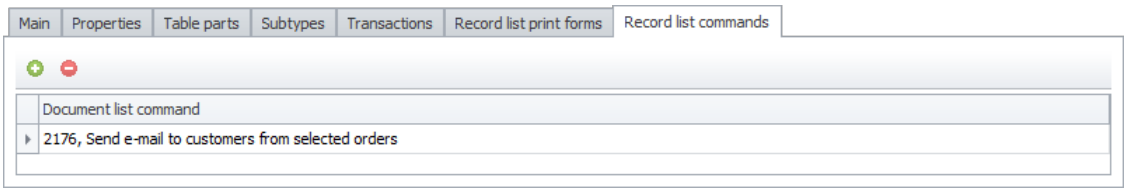
In the "Record list print forms" tab, there is a list of print forms that are available from the list form of documents of such type:



Print forms can be added ⊕ or deleted ⊖ with the help of the corresponding buttons in the tab's toolbar. When adding, a list form of print forms will open, where one can select existing print forms or create new ones. When deleting a print form, it will be deleted only from the list of document's print forms; it will remain in the print forms dictionary. Print forms assigned to a subtype from the print forms edit form will be automatically displayed in this list.

A print form can be opened in the edit form by double left-click on it in the list.

In the "Record list commands" tab, there is a list of commands that are available from the list form of documents of such type:



Commands can be added or deleted with the help of the corresponding buttons in the tab's toolbar. When adding, a list form of commands will open, where one can select existing commands or create new ones. When deleting a command, it will be deleted only from the list of subtype's commands; it will remain in the commands dictionary. A command can be opened in the edit form by double left-click on it in the list. Commands added for a particular subtype via the commands edit form will be automatically displayed in this list.

A command can be opened in the edit form by double left-click on it in the list.

Document class

During creation of each document type, a class of this document is generated. The initial name is represented with document type name (*Name*) +"*Document*", and the list of its properties is represented with corresponding properties of the document type.

For example, let us consider creation of simple document type *DocType* with its properties *AgentID* and *Amount*.

The model class of the subject area, generated according to this description, looks like as follows:

```
public partial class DocTypeDocument : IDocument
{
	public long AgentID { get; set; }
	public decimal Amount { get; set; }

	// System properties created automatically for all types of the documents.
	public bool Deleted { get; set; }
	public long ID { get; set; }
	public long CreatorID { get; set; }
	public string Comments { get; set; }
	public string TotalsList { get; set; }
	public long TypeID { get; set; }
	public long SubtypeID { get; set; }
	public long Version { get; set; }
	public string Description { get; set; }
	public DateTime CreationDate { get; set; }
	public DateTime TransactionDate { get; set; }
}
```

All classes of the documents implement *IDocument interface*. Therefore, a list of all classes of documents can be obtained by requesting who implements this interface:

```
public interface IDocument : IEntity, IBusinessObject
{
	long TypeID { get; set; }
	long SubtypeID { get; set; }
	long CreatorID { get; set; }
	DateTime CreationDate { get; set; }
	DateTime TransactionDate { get; set; }
	string Description { get; set; }
```

```
        string Comments { get; set; }
        string TotalsList { get; set; }
        bool Deleted { get; set; }
        long Version { get; set; }

        // Returns the table parts of the document.
        IKeyValueStore<string, ITablePart> TableParts { get; }
}
```

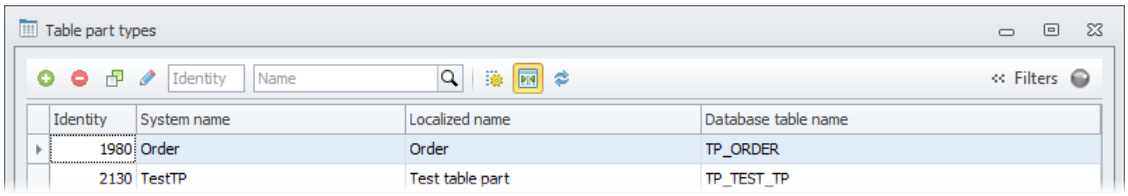
Each document property corresponds the field of *EditableValue<T>* type, where T is one of the types indicated in metadata. A collection of type *DctionaryTable<T> may also correspond to the property* (where *T* – is a type of collection element).

Example of use:

```
[Import]
private IDocumentManager DocumentManager { get; set; }

// Get the document.
var document = DocumentManager<DocTypeDocument>.GetDocument(1488);

// Get the values of document AgentID property.
var agentId = document.AgentID;

// Get the table part of the document.
var tablePart = document.TablePartName;
```

Table parts

 Viewing existing and creating new table parts can be made in the dictionary of table parts Table part types:

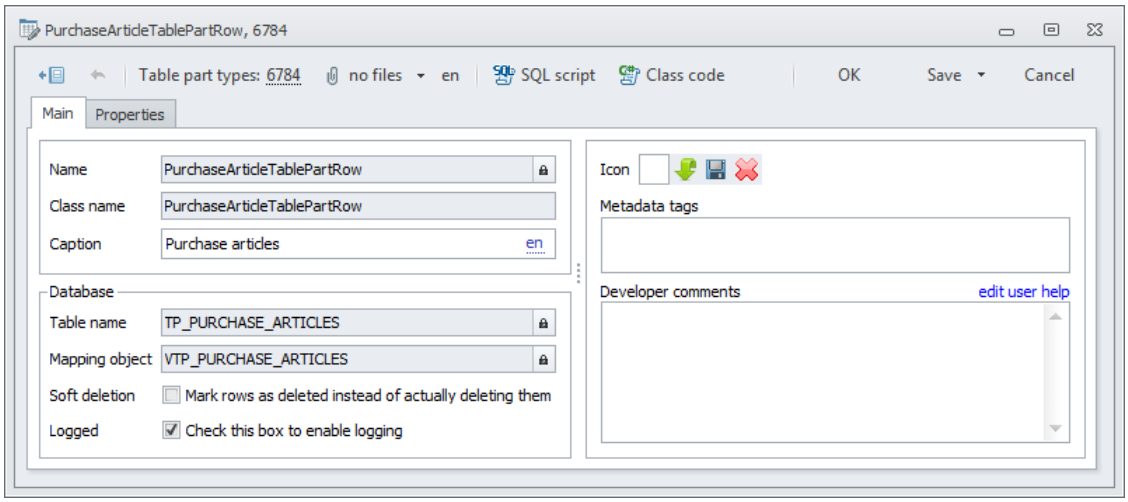


The dictionary records can be filtered by the table part *Name*.

Arrangement of storage for the structure of table parts at the database level is detailed in the section Documents in the chapter KERNEL scheme.

Cloning of metadata objects such as document and table part types is detailed in the Metadata cloning section.

The form for editing of the table part structure contains only two tabs "Main" and "Properties":



The table part class name and its ID are displayed in the form heading.

With the set flag *Soft deletion* (*Database* property group) the table part rows will not be physically removed from the existing table, but will be marked as removed. Activation of this flag will result into the growth of the number of rows in the table, however it facilitates the search over revision history to a great extent. Soft deletion is used generally for table parts of the documents, which content is entered manually by the user. Soft deletion is not recommended to use for table parts of the documents filled in automatically.

Using SQL script button, you can view SQL script for creation or change of the objects of DBMS table part.

Using Class code button, you can view the class generated in C#, describing the table part row.

Tab "Properties" allows enumeration of the table part fields similarly to the fields of document head. Multilanguage feature is not supported for the table parts.

The table part is assigned to the document in the form for editing of the document type in the tab "Table parts".

Class of table part record

During creation of each table part, a class of table part record is generated. Its initial description is represented with table part name (*Name*) +"*TablePartRow*", and the list of its properties is represented with corresponding properties of the table part.

For example, let us consider creation of simple table part*DocumentName* with *ArticleID* and *Amount* properties.

The model class of the subject area, generated according to this description, looks like as follows:

```
public partial class DocumentNameTablePartRow : ITablePartRecord,
{
        public long ArticleID { get; set; }
        public decimal Amount { get; set; }

        // System properties created automatically for all types of the documents.
        public long ID { get; set; }
        public long DocumentID { get; set; }
        public long TablePartEntryID { get; set; }
        public bool Checked { get; set; }
        public DateTime TransactionDate { get; set; }
```

```
        public bool Deleted { get; set; }
        public bool DocumentDeleted { get; set; }
}
```

All classes of records of the link tables implement the interface *ITablePartRecord*. Therefore, a list of all classes of records of the link tables, can be obtained by requesting who implements this interface:

```
public interface ITablePartRecord : IEntity, IBusinessObject
{
        long DocumentID { get; set; }
        DateTime TransactionDate { get; set; }
        bool DocumentDeleted { get; set; }
        bool Deleted { get; set; }
        long TablePartEntryID { get; set; }
        bool Checked { get; set; }
}
```

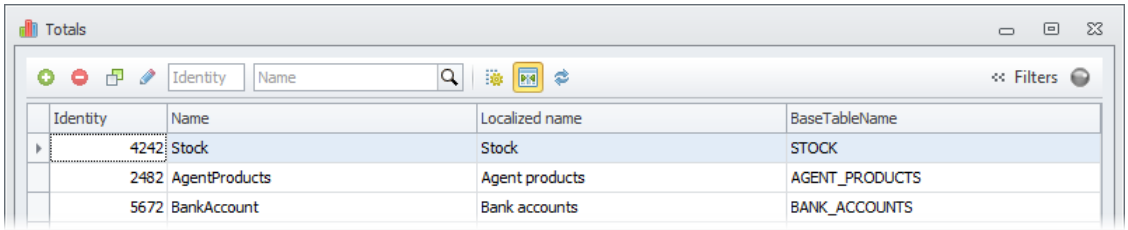
The field of type *EditableValue<T>* corresponds to each property of table part record, where T is one of types indicated in metadata:

Example of use:

```
[Import]
private ITableSourse DataContext { get; set; }

var documentId = 100500;
var deleted = 0;

var query =
        from tablepart in DataContext.GetTable<DocumentNameTablePartRow>()
        where tablepart.DocumentID == documentId &&
                tablepart.Deleted == deleted
        select tablepart;

return query.ToDictionary(
        tablepart => tablepart.ArticleID, tablepart => tablepart.Amount);
```

Totals

You can view the existing and create new totals in the Totals dictionary:

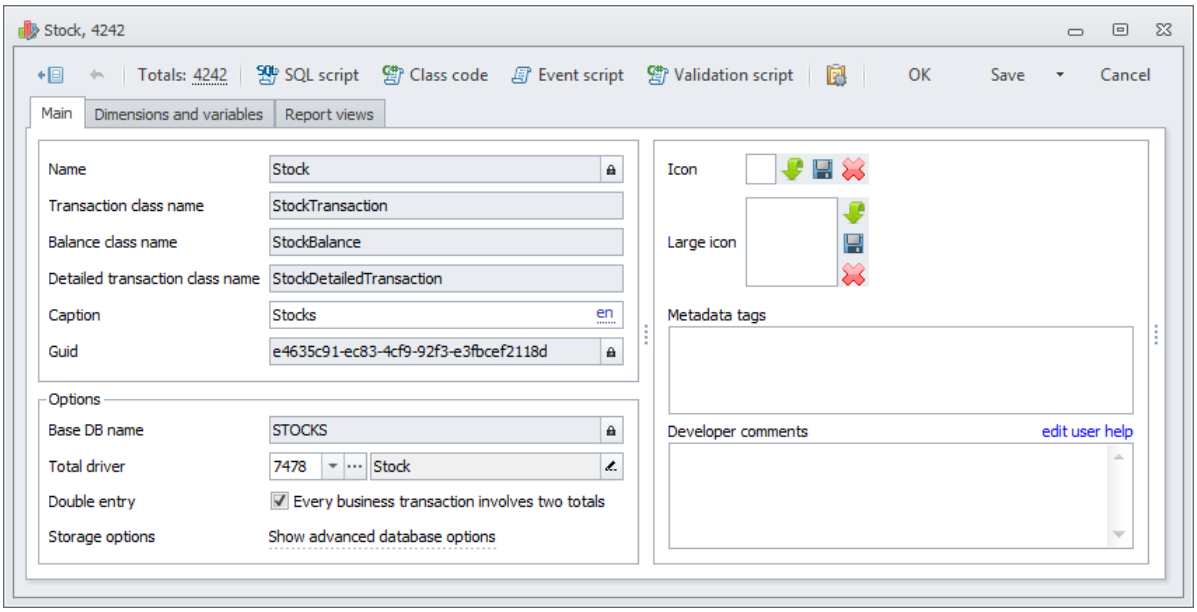


The dictionary records can be filtered by *Name* of Totals (*Name*).

Arrangement of storage for the structure of totals at the database level is detailed in the section Totals in the chapter KERNEL scheme.

Cloning of metadata objects such as dictionaries, link tables, and totals is detailed in the Metadata cloning section.

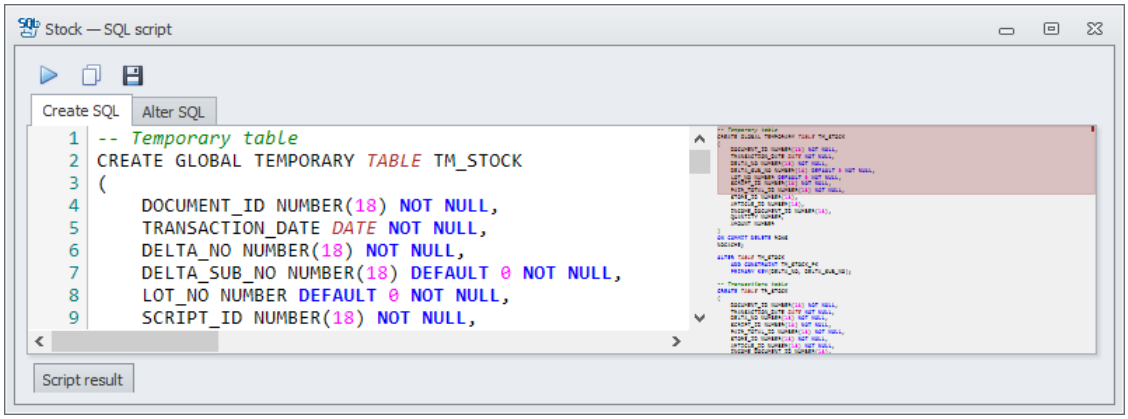
Properties of Total in structure editing form grouped by tabs for convenience:



The Total class name and its ID are displayed in the form title.

The following buttons exist in the form tool panel besides total ID (automatically set):

SQL script is script for using in any application for work with report data base, supporting SQL (PL SQL Developer, TOAD etc.) for creation Totals object in following data base:



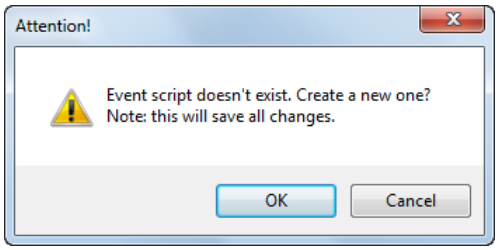
- Script for creation Total objects in data base is located in the "Create SQL" tab. This script can be used if the object creates in SQL firstly;
- Script for deleting the Total objects in data base is located in the tab "Alter SQL".
- Buttons on the panel on the top side of form allow:
  - - do current script in data base,
  - - copy current scrip to exchange buffer,
  - - save current script to the file on disc.

Script which is used for change created total objects in data base, is not generate by system.

The reason is deficiency of secure total change technique on operating configuration. Change of total is possible only at the stage of implementation by deleting tables and recreating them. After implementation you can only create a new total and replace the old. For reporting purses data of two totals co-operate by creating user report virtual total.

Class code – class, which describes the total generated in C#;

Event script – script of total events handler . During creation of a new total, the events handler is not created. The system offers to create it when the button firstly pressed. All changes will be saved in this case.

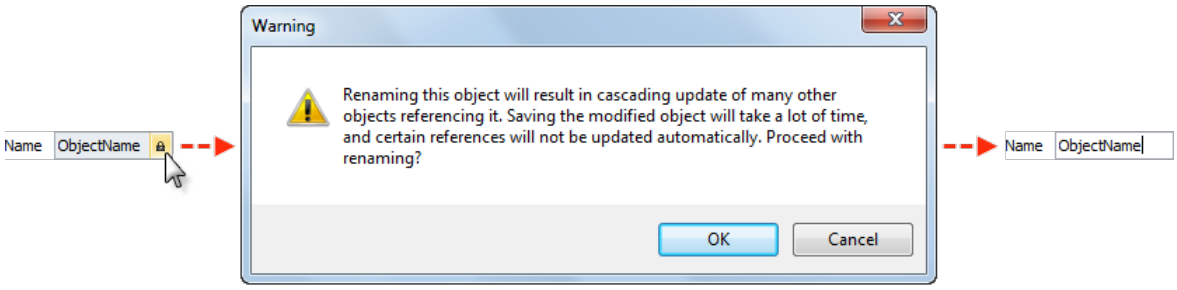


Validation script – script transaction validator. During creation of new total, the validator script is not created. The system offers to create it whet the button held firstly. All changes will be saved in this case.

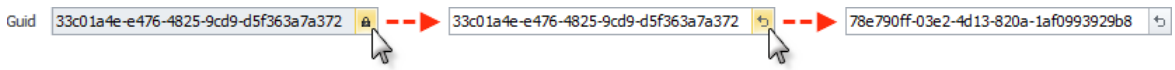
- button of opening editing total report parameters form (functional operates only for created totals).

In the tab "Main" main total properties are organized:

- *Name* – total name. It must accord to the name of one total motion (must be in the singular). In the example below it is the total name *rummage on stock* – *Stock*. If necessary, it can be changed:



- *Transaction class name* – total transaction class name Is generating automatically based on *Name* total ;
- *Balance class name* – total balanse class name Is generating automatically based on *Name* total ;
- *Detailed transaction class name* – total detalised transaction class name Is generating automatically based on *Name* total ;
- *Caption* – total name which displayed on screen forms (for example, report parameters form) Is generating automatically based on *Name* total ; For example, for total with name *BankAccount* this property will has the value *Bank accounts*;
- *Guid* – is used to identify a menu item.
  Guid is generated automatically at random and, if necessary, (in case of coincidence with Guid of another object) can be changed:



- *Base DB name* – is basic name of total object in *application scheme* the data base. On it base names of other objects - tables and presentations - create by adding prefix Basic name is generated automatically on the base of name *Name* of total and can be changed in case of necessity:

  Name can contain only Latin letters, figures and sign "_". The name must begin with a letter and the length can't be more than 30 symbols (available amount of symbols are shown on control element);
- *Total driver* – total driver
- *Double entry* – flag of double record which use for balanse total If the flag set the rule of double entry will be applied in case any changes.

The double entry rule means that amount in column *Amount* of totals of two transactions (or wire) must be is equl to zero. So the balance of all column total *Amount* always is equal to zero. The rule can be check in two places:

- While saving operation transactions during document saving;
- While saving calculated transactions after handling by total driver.

Mistake in total driver can cause divergence of transactions balance amounts. In this case calculations ends, and comply message is recorded to the server log;

- *Use trigger-based balance table* — flag shows the necessity of balance tables using (TB_ and TT_ accordingly for operating and analytic parts). In case the flag isn't set, the total balance tables will not be created. This option is only displayed when the *Show advanced database options* link is clicked.

  Balance tables — one of the main reason for deadlocks while document running. However, refusal from balance table greatly increases the time for making report, if the user query incoming and outgoing variable values. Besides it, many of totals need the control of total values. For example, rummage on stock have limitations ate the balance table, which prevent decrease the goods amount below zero. The good candidate for total which can be used without balance tables is realization total. The user usually interested in the period total (without incoming and outgoing values);
- *Icon* – standard icon (with the size of 16 x 16 pixels).

  Icons are shown, for example, on the main menu or at the list form title name and dictionary editing form.

  The buttons to the right of icon preview area allow:

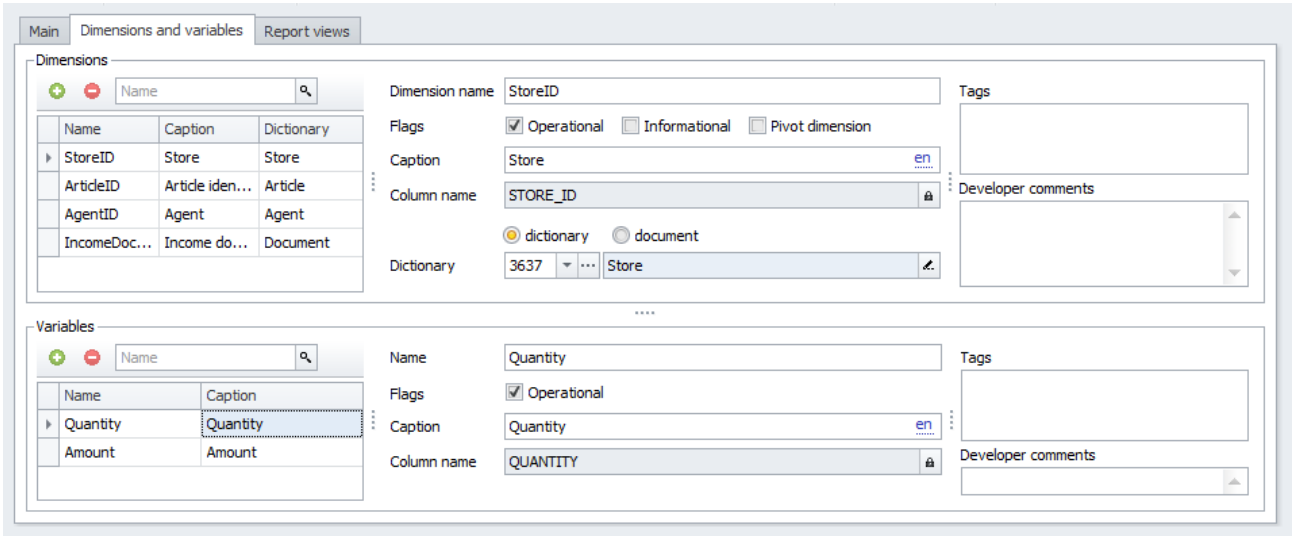
  – loading the icon;

  – saving the icon previously downloaded to the computer;

  – deleting the icon;
- *Large icon* – a large icon (with the size of 32 x 32 pixels);
- *Metadata tags* – tags used to describe total functionality; Used for searching the objects implemented for certain functionality associated with such tag.

  The tag is added by keys Space or Enter. deleted –by button after the tag. As the space is used for tag entering you can replace by signs "_" or "-" in tags with the name of several words;
- *Edit user help* – comment to the object which the end user can see in the form of a hint which drops down after mouseover. The comment is entered for any language of the system;
- *Developer comments* – comments of the application developer;

On the top of the lap "Dimensions and variables" can be found the total dimensions list *Dimensions* and on the bottom is situated the list of variables *Variables*. On the left side of each list is situated the properties list (dimensions and variables), on the right side - all parameters of the property selected from left:



The dimensions and variables can be added or removed using corresponding buttons in the toolbar: They also can be filtered by the name (*Name*).

Each dimension has:

- *Dimension name* – the name which automatically acquires end *ID*;
- *Operational* – a flag indicating if the dimension is operational. In case the flag is set, the value of operational dimension will be formed in the result of the work of transaction scripts while saving documents. In case the flag isn't set, the value of (analytic) dimension will be calculated by total driver;
- *Caption* – a name displayed in the screen forms; It is generated automatically on the base of the dimension name *Name* by dividing into words;
- *Column name*- the name of corresponding field in total tables in *application scheme of* data base . The name is generated automatically. It can be changed in the same way as the base name of total objects in data base in case of necessity. Field name has the same limitations as the names of other data base objects.
- *Dictionary/Document* - the dictionary or document, elements of which are the value of dimension;
- *Tags* – tags used for description of the property functionality. It is analogue in the meaning to total tags;
- *Developer comments* – comments of application developer;

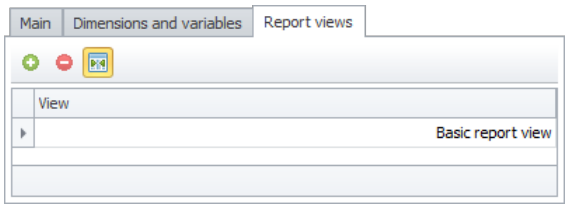
All total dimensions have the data type *long*, all variables – *decimal*.

Two variables are automatically created while new total creating *Quantity* (quantity) and *Amount* (amount). They the most frequently used total dimensions. They can be deleted if they are no longer needed.

Amount – reserved dimension name, used for the control of the double entry rule. Any total with the control of the double entry rule (set by flag *Double entry*) must have the column with name *Amount*.

The list of columns available on default is located in the lap "Report view" and they used to form the column list during the creating of report.

Column lists can be added or removed with corresponding buttons in the toolbar.



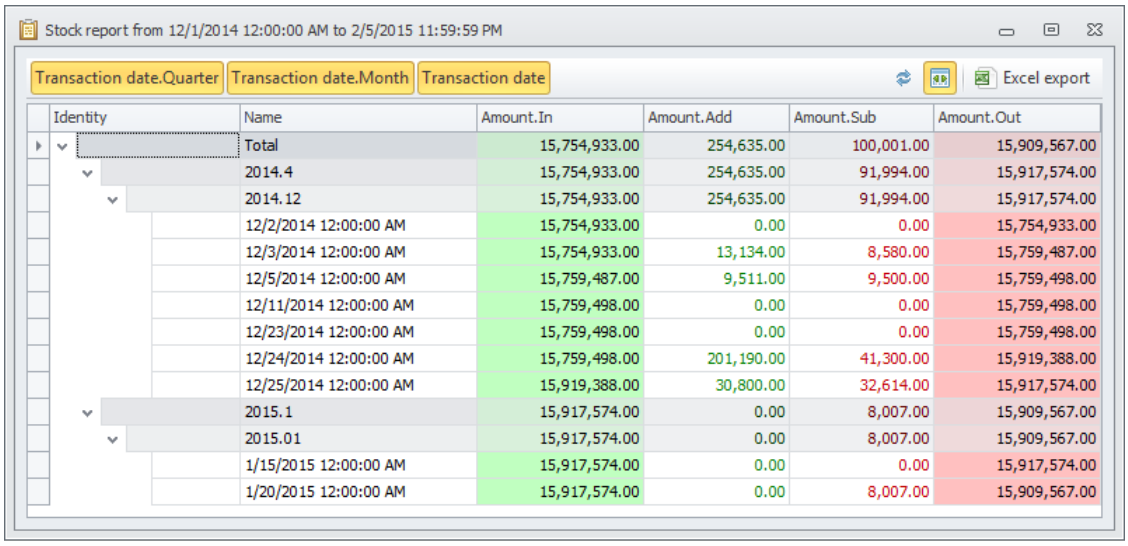
Total data is calculated regularly and automatically by activity "Total calculation".

Reports on the totals

There are three types of reports in the system:

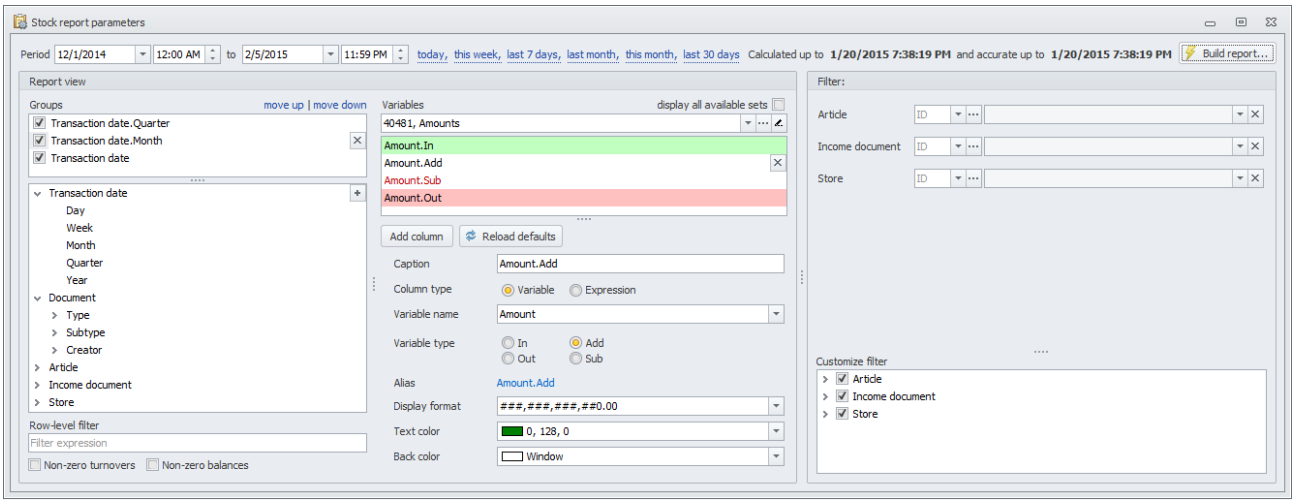
- reports on the totals, which functionality is implemented completely with the kernel ;
- virtual totals, using which a report can be prepared on the consolidated data of several totals;
- user reports, implemented by the application developer.

The functionality of the reports implemented with system and available for the totals provides a user with flexible mechanism of report data handling. The same functionality is used in the virtual totals and user reports:



Identity	Name	Amount.In	Amount.Add	Amount.Sub	Amount.Out
	Total	15,754,933.00	254,635.00	100,001.00	15,909,567.00
	2014.4	15,754,933.00	254,635.00	91,994.00	15,917,574.00
	2014.12	15,754,933.00	254,635.00	91,994.00	15,917,574.00
	12/2/2014 12:00:00 AM	15,754,933.00	0.00	0.00	15,754,933.00
	12/3/2014 12:00:00 AM	15,754,933.00	13,134.00	8,580.00	15,759,487.00
	12/5/2014 12:00:00 AM	15,759,487.00	9,511.00	9,500.00	15,759,498.00
	12/11/2014 12:00:00 AM	15,759,498.00	0.00	0.00	15,759,498.00
	12/23/2014 12:00:00 AM	15,759,498.00	0.00	0.00	15,759,498.00
	12/24/2014 12:00:00 AM	15,759,498.00	201,190.00	41,300.00	15,919,388.00
	12/25/2014 12:00:00 AM	15,919,388.00	30,800.00	32,614.00	15,917,574.00
	2015.1	15,917,574.00	0.00	8,007.00	15,909,567.00
	2015.01	15,917,574.00	0.00	8,007.00	15,909,567.00
	1/15/2015 12:00:00 AM	15,917,574.00	0.00	0.00	15,917,574.00
	1/20/2015 12:00:00 AM	15,917,574.00	0.00	8,007.00	15,909,567.00

The total dimensions act as the objects *Groups*, according to which grouping of report data is carried out:



The Column providers are used to create additional levels of details (grouping).

There are initially two Column providers in the system by default: documents and dictionaries:

- the provider of documents provides a possibility to detail the reports by the attributes of the very documents Document and time periods Transaction Date;
- the provider of dictionaries provides a possibility to give additional details for the reports by the values of properties-links of the dictionaries.

If this level of details is insufficient for some dictionary being the total dimension, own Column providers can be created for it.

The total variables act as the objects *Variables*, using which the report columns are built (except for the columns *Identity* and *Name*). The columns can be selected either with a set from preliminary built report types, or formed independently.

Total transaction class

During creation of each total, three classes of total transaction are generated. Their initial description is represented with the total name (*Name*) +"*Transaction*" for the total transaction class, +"*Balance*" for the balance transaction class and +"*DetailedTransaction*" for the total detailed transaction class. The initial description of the list of their properties is represented with corresponding dimensions and variables of the total.

For example, let us consider creation of simple total *TotalName* with dimension *DimensionID* and variable *Variable*.

The model class of the subject area, generated according to this description, looks like as follows:

```
// Total transaction class.
public partial class TotalNameTransaction : TransactionValue
{
	public long DimensionID { get; set; }
	public decimal Variable { get; set; }
}

// Total balance class.
public partial class TotalNameBalance : BalanceValue
{
	public long DimensionID { get; set; }
	public decimal Variable { get; set; }
}

// Total detailed transaction class.
public partial class TotalNameDetailedTransaction : DetailedTransactionValue
{
	public long DimensionID { get; set; }
	public decimal Variable { get; set; }
}
```

The classes of of totals' transactions are inherited from the following base classes:

- the classes of totals' transactions are inherited from the class *TransactionValue*, which is inherited in turn from the base class *TransactionBase*:

```
public abstract class TransactionValue : TransactionBase,
	IEquatable<TransactionValue>, IComparable<TransactionValue>
{
	public long DocumentID { get; set; }
	public DateTime TransactionDate { get; set; }
	public long DeltaNo { get; set; }
	public long ScriptID { get; set; }
	public long PairTotalID { get; set; }
	public Document Document { get; set; }
	public Total Script { get; set; }
	public Total PairTotal { get; set; }
}
```

- the classes of totals' balances are inherited from the class *TransactionValue*, which is inherited in turn from the base class *TransactionBase*:

```
public abstract class BalanceValue : TransactionBase, IEquatable<BalanceValue>
```

- the classes of totals' detailed transactions are inherited from the class *DetailedTransactionValue*, which is inherited in turn from the base class *TransactionBase*:

```
public abstract class DetailedTransactionValue : TransactionValue,
       IEquatable<DetailedTransactionValue>, IComparable<DetailedTransactionValue>
{
       public long DeltaSubNo { get; set; }
       public decimal LotNo { get; set; }
}
```

Each total transaction property corresponds to the field of *EditableValue<T>* type, where T is one of the types indicated in metadata.

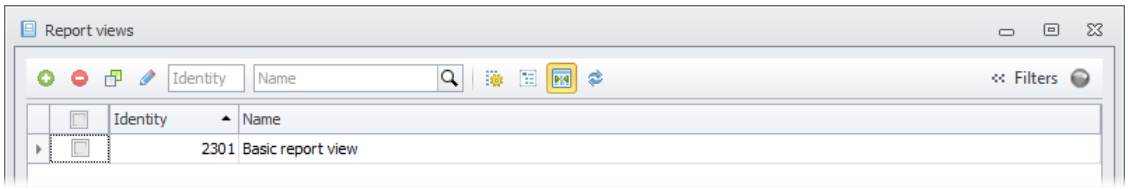
Example of use:

```
[Import]
private ITableSourse DataContext { get; set; }

var documentId = 100500;

var query =
       from total in DataContext.GetTable<TotalNameTransaction>()
       where total.DocumentID == documentId
       select total;

return query.ToDictionary(
       total => total.DimensionID, total => total.Variable);
```

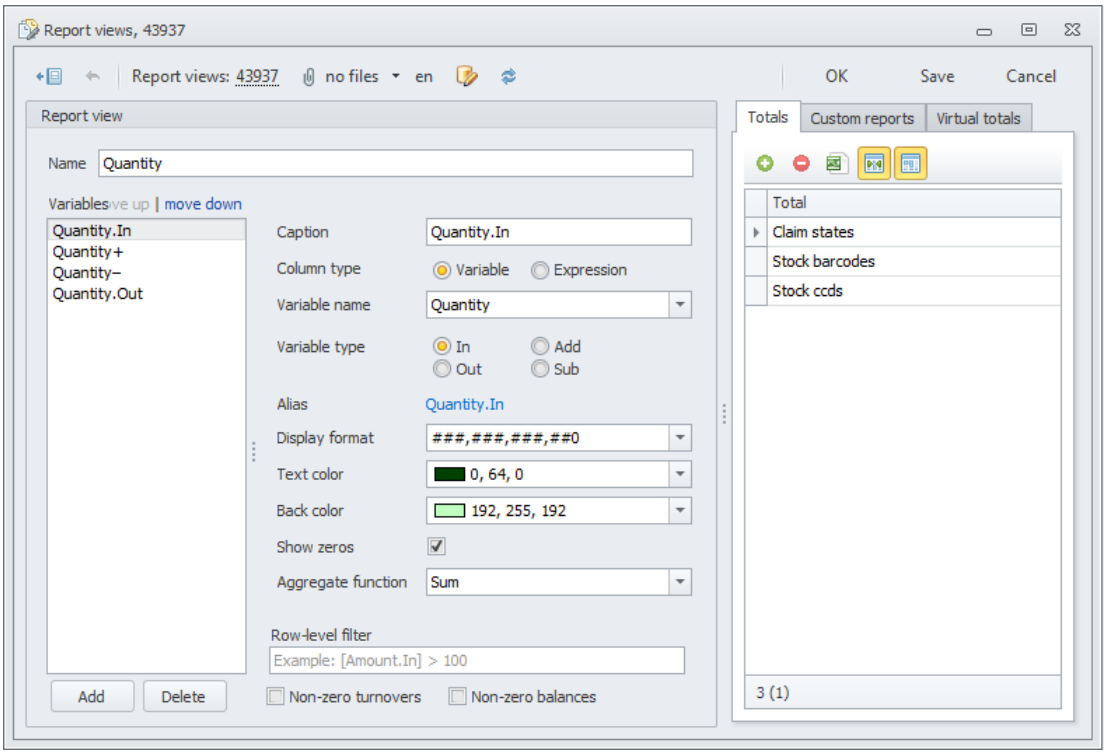
Report types

Report types serve to provide the opportunity for rapid choice of report column set while parameters are change-settings.

The list of all column set can be found in the dictionary “Report views":

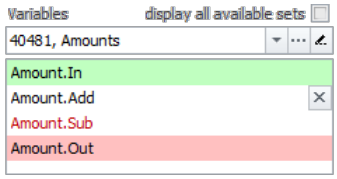


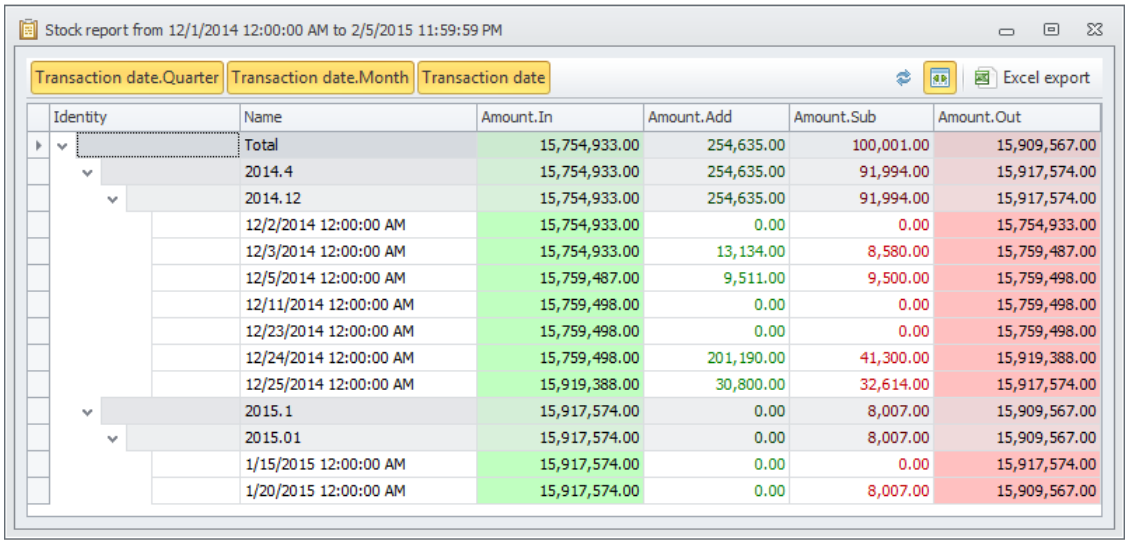
The properties of column set in the editing form grouped on the following way for convenience:



Properties of report type is shown on the “Report type setting” group:

- *Name* - name;
- Report type column list is located on the left side of group, and properties of selected column is on the right side. The order of column in the list (from up to down) accord to the order of it displaying in the report on default (from left to right):





Identity	Name	Amount.In	Amount.Add	Amount.Sub	Amount.Out
	Total	15,754,933.00	254,635.00	100,001.00	15,909,567.00
	2014.4	15,754,933.00	254,635.00	91,994.00	15,917,574.00
	2014.12	15,754,933.00	254,635.00	91,994.00	15,917,574.00
	12/2/2014 12:00:00 AM	15,754,933.00	0.00	0.00	15,754,933.00
	12/3/2014 12:00:00 AM	15,754,933.00	13,134.00	8,580.00	15,759,487.00
	12/5/2014 12:00:00 AM	15,759,487.00	9,511.00	9,500.00	15,759,498.00
	12/11/2014 12:00:00 AM	15,759,498.00	0.00	0.00	15,759,498.00
	12/23/2014 12:00:00 AM	15,759,498.00	0.00	0.00	15,759,498.00
	12/24/2014 12:00:00 AM	15,759,498.00	201,190.00	41,300.00	15,919,388.00
	12/25/2014 12:00:00 AM	15,919,388.00	30,800.00	32,614.00	15,917,574.00
	2015.1	15,917,574.00	0.00	8,007.00	15,909,567.00
	2015.01	15,917,574.00	0.00	8,007.00	15,909,567.00
	1/15/2015 12:00:00 AM	15,917,574.00	0.00	0.00	15,917,574.00
	1/20/2015 12:00:00 AM	15,917,574.00	0.00	8,007.00	15,909,567.00

  - links up and down move selected column on the list level up or down consequently;
  - The button Add adds new column to the list;
  - The button Delete deletes selected column from list;
- Title - the name of list shown in report ;

- *Type* – column type :
  - *Variable* – means that the mean can be taken from data;
  - *Expression* – the value will be calculated in accordance with formula;
- The following parameters are available for Variable column type :
  - Variable name - the name of total variable, the value of which will be displayed in the column. Variable name must coincide to the name of Variable of total ;
  - Flags (mutex) which indicate variable values stated in the column:
    - Starting balance;
    - Ending balance;
    - Incoming;
    - outgoings;
  - Alias - the column alias which is used in expressions (Expression column type). The left mouse button click copies it to the exchange buffer;
- The following parameters are available for Expression column type:
  - Column name - the unique column name used for calculation of report data (alias analogue for variable column type);
  - Expression - expression which forms column value (described in detailed hereafter);
- Format - format which shows values in column (type of all report columns *Decimal*). Format can be selected from predefined by pressing control element or write by user. Detailed description of standard number format can be found on MSDN eng/rus. Detailed description of customisable number format can be found there eng/rus;
- *Text color* – column text color;
- *Background color* – column background color;
- *Show zeroes* — if the cell has 0 value and ticked the box 0 will be displayed, if the box is unticked - the cell will be empty;
- Aggregate function - the function which used for subtotal in report form on default.

In writing the expression forms column value is available following opportunities:

- As expression of variable can be used alias *Alias* of column type *Variable* (of the same report type) enclosed to square parenthesise:

```
[ColumnName] / 100
```

- Result report table contains system variable *GroupLevel* – group level in report tree (the first line*Total* has level 1), and *GroupCount* – total amount of levels on the tree, which can be used in creating the expressions with ternary operator ?:

```
[GroupLevel] == [GroupCount] ? min([ColumnName]) : max([ColumnName])
```

- Keeps the numbers with floating point – 1.5E5;
- Keeps types – true, false;
- Keeps system functions from System.Math namespace:
  - *Abs* – absolute number value;
  - *Max* – returns the larger of two numbers:

```
Max([ColumnName1], [ColumName2])
```

  - *Min* – returns the smaller of two numbers;
  - *Pow* – returns the number raised to the stated power:

```
Pow([ColumnName], 2)
```

  - *Round* – rounds off the number to the nearest whole number;
  - *Round* – rounds the number to the stated number of divisional class:

```
max([ColumnName])
```

  - *Sqrt* – the square root of;
  - The totallist of functions can be found at MSDN eng/rus;

- Supports additional functions:
  - *zdiv* – secure division with stated result (third parameter), if the divisor is equal to zero:

```
zdiv([ColumnName1], [ColumnName2], 0)
```

  - Aggregate functions which calculate the value of lines in own group (on the branch of report tree):
    - *avg* – average value;
    - *count* – total lines amount in the group;
    - *max* – maximum value;
    - *min* – minimum value;
    - *sum* – summary value.

```
sum([ColumnName])
```

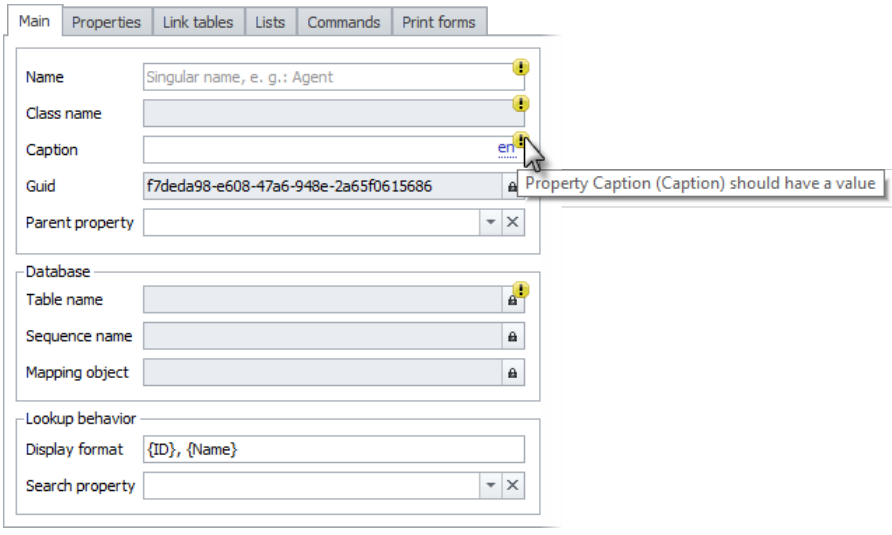
the totals enumerates in the “Totals” tab and this column set will be available on default in the report parameters editing form. As every report columns set can be defined for several totals, so every total can be defined for several column set. Totals in list can be added or removed with corresponding buttons in the panel.

user reports enumerates in the “User reports” tab and this columns set will be available on default in the parameters setting form. As every report columns set can be defined for several user reports, so every user report can be defined for several column set. The user reports in list can be added or removed with corresponding buttons in the panel.

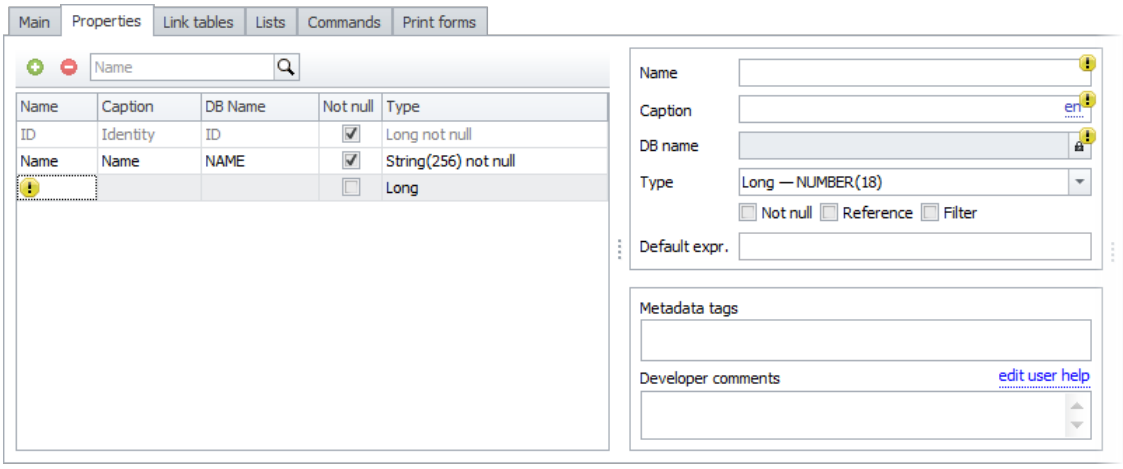
Virtual totals enumerates in the “Virtual totals” tab and this columns set will be available on default in the parameters setting form. As every report columns set can be defined by several virtual totals, so every virtual total can be defined for several column set. Virtual totals in the list can be added or removed with corresponding buttons in the toolbar.

Metadata Validation

The forms for creation of metadata objects support validation of created objects. After mouseover at the error icon, a validator hint appears:

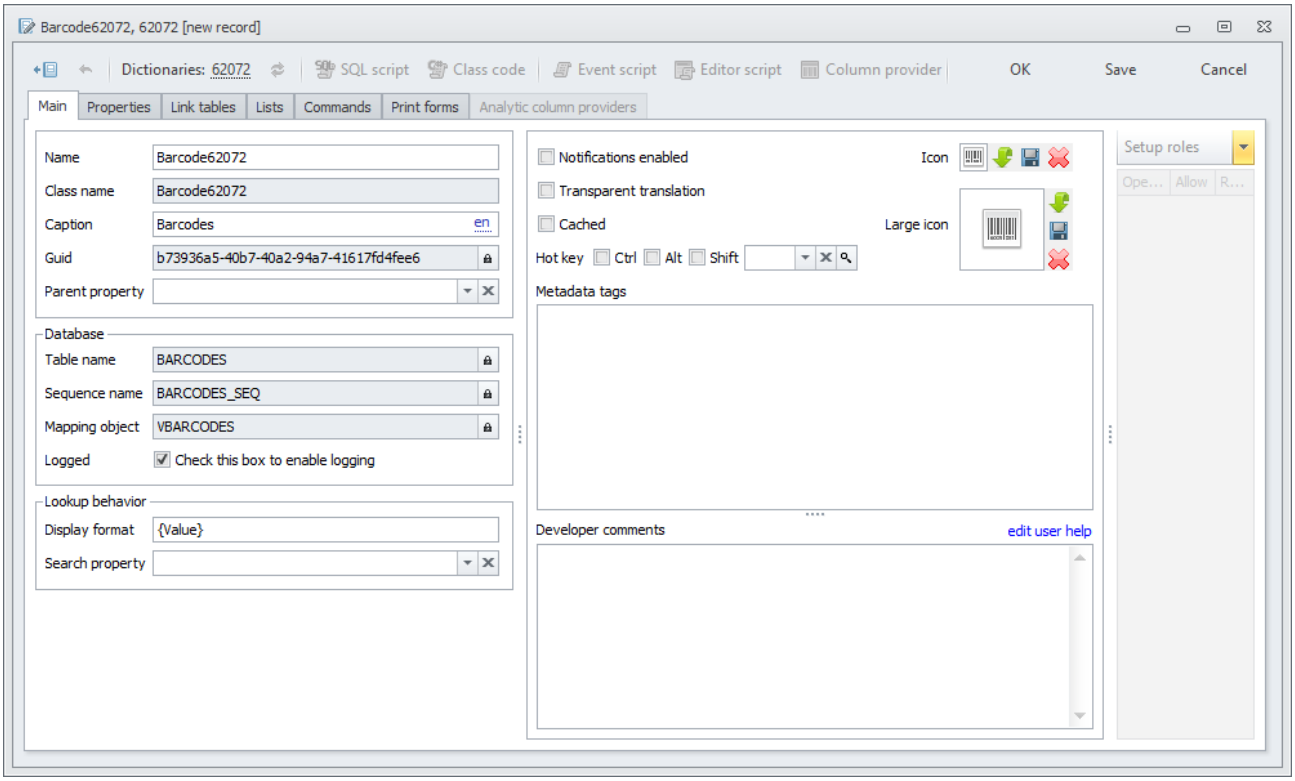


If the element displayed in the table contains an error, the error icon is displayed directly in the table in the row containing the element:



Metadata Cloning

Cloning of the metadata objects helps creating new objects of the similar structure. Like with cloning of the ordinary dictionaries and documents, an existing record is loaded and used as the template for the new record displayed on the screen and presented to the user allowing to modify the object before saving (rename, add comments, modify or add new properties, etc). The object is not saved to the database until the user clicks the Save button:



Metadata objects define the structure of the application classes and data tables, so the metadata cloning has its specifics:

- Cloned objects are given unique names (numeric suffix is automatically appended to the new object's name) to make sure that class names don't conflict with the original objects;

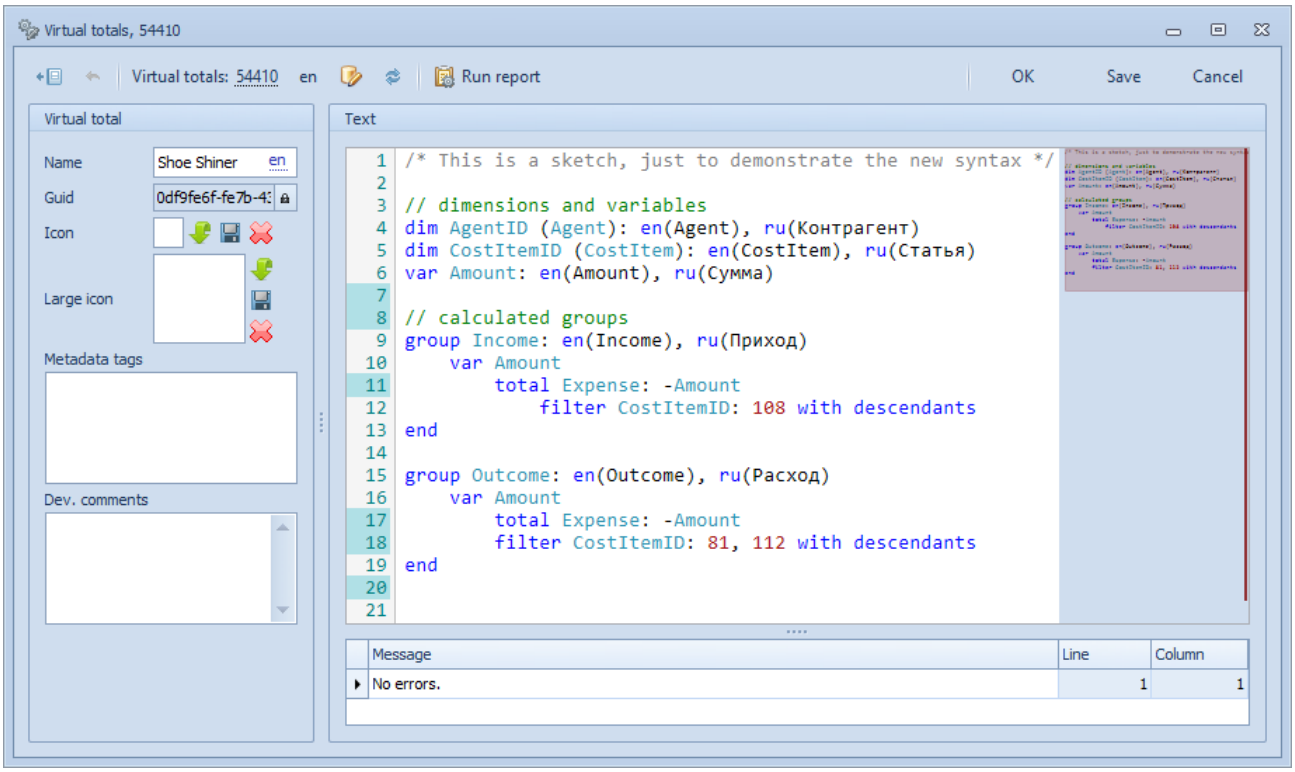
- Inner objects and script references are cleaned up to make sure that the clones don't share their internals with the original objects;
- Dictionary specifics:
  - Event handler, editor event handler and report column provider scripts are deleted;
  - Inner dictionaries are deleted (because they reference the original dictionary, not the clone anyway);
  - Inner link tables are deleted (because they reference the original dictionary, too);
  - Dictionary constants are deleted (because the new dictionary will have its own data).
- Document type specifics:
  - Event handler, editor event handler and document transaction scripts are deleted;
  - Subtypes and table parts are deleted (because using the same table parts types in different document types is strictly discouraged).

All other properties of the cloned objects, including the database table names, are kept as is. It is up to the developer to make sure that the cloned object works as expected: the clone requires the same debugging and testing cycle as any newly created metadata object.

Virtual totals


Virtual totals are a technical name of a financial statement mechanism. They allow combining data of several totals and bringing them in one summary report. For a real user reports on totals and on virtual totals look like exactly similar: the same opportunities of filtering, group, detailing, variables and expressions selection. Unlike user reports having similar potential, virtual totals are much easier in setup and are oriented on business analysts but not on programmers.

On the engineering side the virtual totals are a source of analytical data, similar to normal totals but created on the fly according to the given rules. Reports on the virtual totals are built by means of the standard mechanism of reporting, and rules of data collection are formulated in special language which for a nonexpert is easier to master, than SQL:



Finally data source for any virtual total are tables of normal totals. Data for off-balance totals are always undertaken from operational tables, for totals with double record are usually from analytical.

Exact rule is: if in the virtual total all dimensions and variables (including involved in filters and predicates) are operational, then for totals with double record data are undertaken from operational tables. If at least one dimension or variable is analytical that data for all totals with double record are undertaken from analytical tables. This rule guarantees mutual coherence of data with which the virtual total operates.

The virtual total constructed according to operational tables always shows digits that are actual at the time of the report calculation. If analytical tables are used in it, then its relevance depends on till what moment the balance totals which are in its part are calculated. In typical cases the actual point is in limits of hour from the present situation.

Structure overview

The virtual total consists of the following elements:

- Dimension — as a usual total has, the reference to dictionaries or documents
- Variables — as a usual total has, numerical data for summation (group)
- Calculated groups — rules for summation of the data, obtained from several sources
  - Sources-totals — rules for extraction of data from typical total
  - Sources-groups — rules for extraction of data from other calculated groups
  - Filters — expressions for allocation of a subset of the necessary data
- Parent groups — rules for the organization of the calculated groups into the hierarchy
- Predicates — external filters, which can not be formulated within language of virtual totals

Commands for reports calculation on them are automatically formed for all virtual totals in the system. Commands names are formed on the basis of names of virtual totals and localized similar to the names of totals and dictionaries.

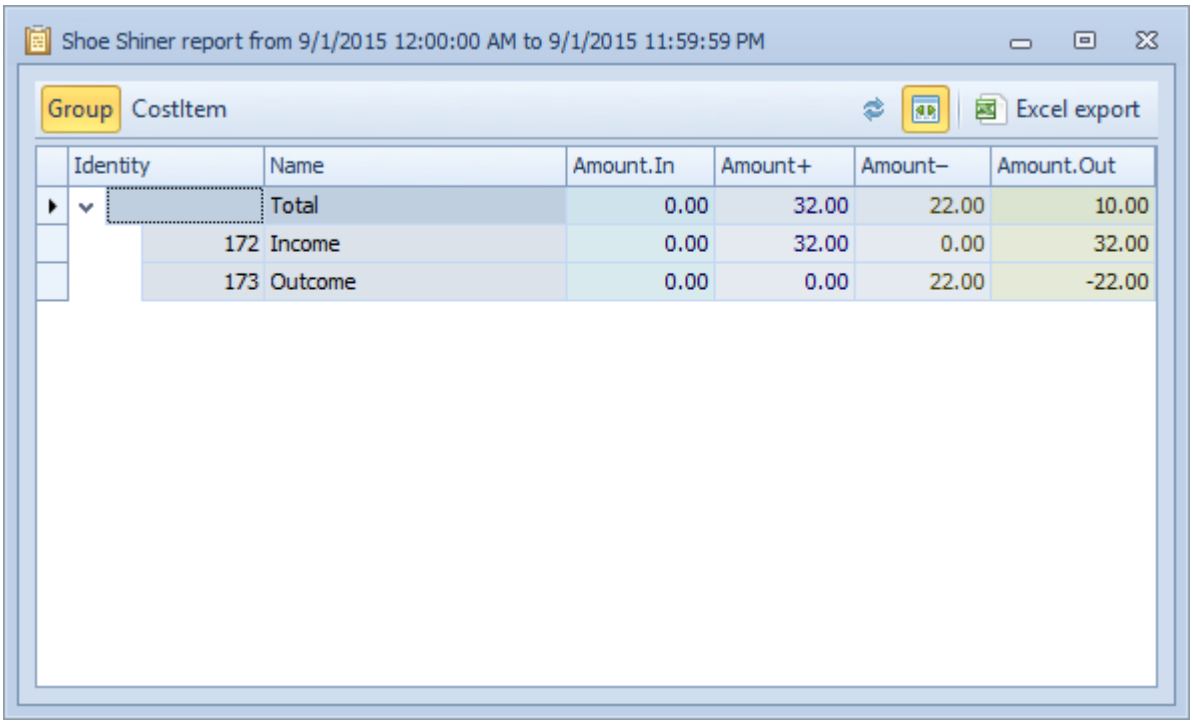
Example: profits and losses

For complete description of virtual total it is enough to specify a list of dimensions, variables and calculated groups (remaining components are optional). dimensions and variables have the same sense that is in normal totals. Calculated groups are main mechanism that makes virtual totals by such powerful analytical tool. It allows integrating and filter data from several totals or other calculated groups in one line of the report. We will consider it on the most simplified sample of profit and loss report.

A shoe polisher in hotel provides his simple services to guests and accepts arbitrary currency which he changes for rubles at a current rate in an adjacent exchanger of currency as payment. The list of his services is fixed: brushing with shoe-polish, washing by a rag in a bucket with water and processing of footwear with a deodorant. Production costs: rent of the place in the hall, shoe-polish, a deodorant, brushes and rags. To consider all this, there would be enough one Cost total for him if not fuss with currency. For loss record because of currency exchange one more total will be required: Conversion.

Costs and Conversion can be integrated in one report using the virtual totals. And to separate incomes from outcomes calculated groups are used:

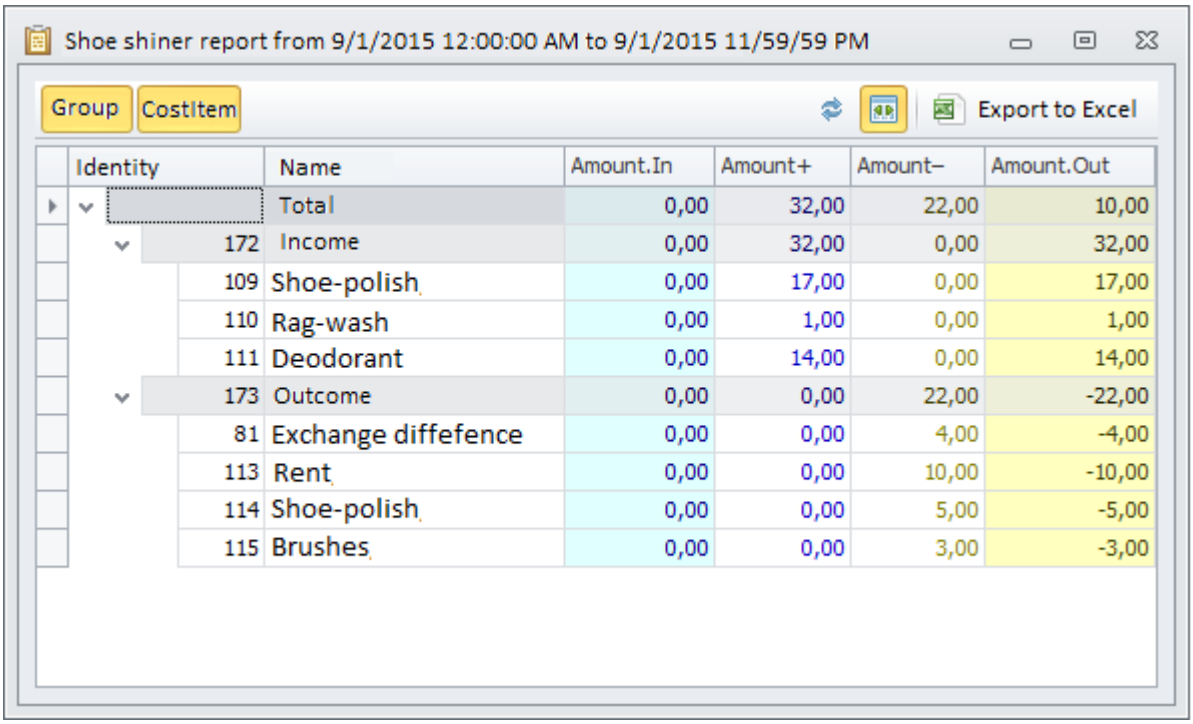
- Calculated Income group: Cost total (we take only Shoe-polish, Washing, Odor removal articles)
- Calculated Outcome group: Cost total (Rent, Shoe-polish, Brushes, Rags, Deodorant articles) + total Conversion
- Total: Profit group + Losses group



Identity	Name	Amount.In	Amount+	Amount–	Amount.Out
	Total	0.00	32.00	22.00	10.00
172	Income	0.00	32.00	0.00	32.00
173	Outcome	0.00	0.00	22.00	-22.00

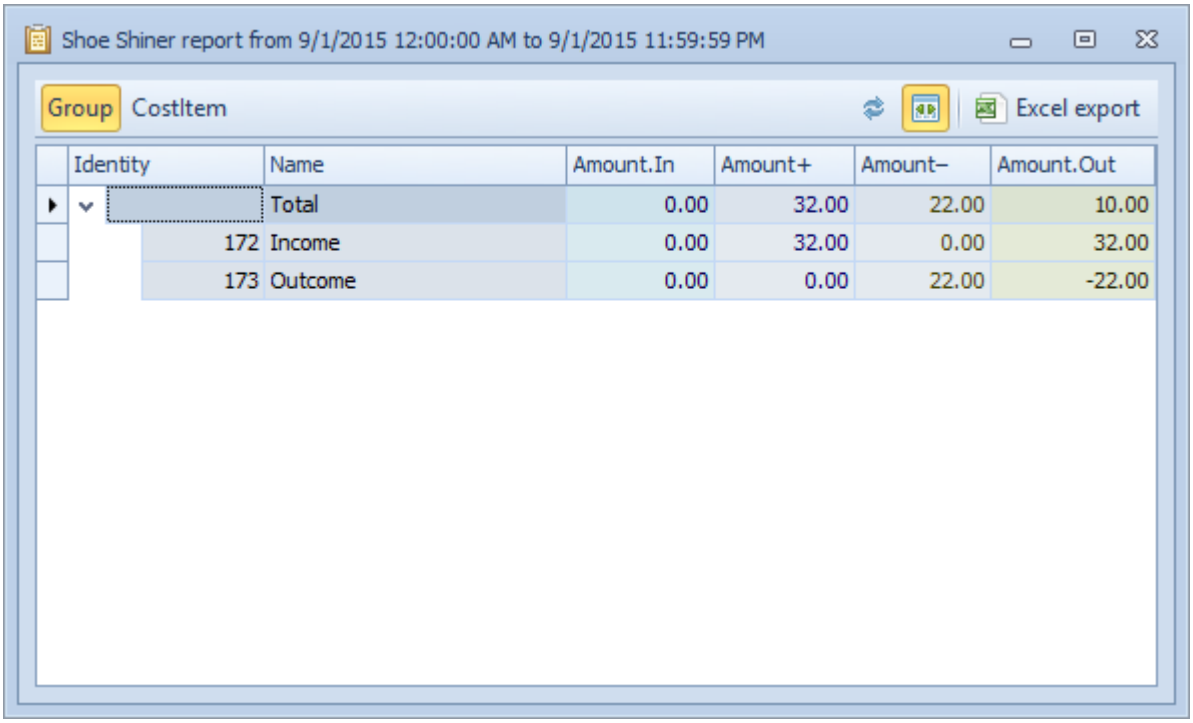
When our hypothetical shoe polisher is checked with his report on a brand new pad, he sees three lines: Income. Outcome, Total. Besides, as much as detailed refining is available to him for all these lines — whereof today's profit specifically is combined, what outcomes were, how many services were

rendered, how many brushes are acquired and how much money is lost because of exchange difference in case of exchange of dollars for rubles:



Identity	Name	Amount.In	Amount+	Amount–	Amount.Out
	Total	0,00	32,00	22,00	10,00
172	Income	0,00	32,00	0,00	32,00
109	Shoe-polish	0,00	17,00	0,00	17,00
110	Rag-wash	0,00	1,00	0,00	1,00
111	Deodorant	0,00	14,00	0,00	14,00
173	Outcome	0,00	0,00	22,00	-22,00
81	Exchange diffefence	0,00	0,00	4,00	-4,00
113	Rent	0,00	0,00	10,00	-10,00
114	Shoe-polish	0,00	0,00	5,00	-5,00
115	Brushes	0,00	0,00	3,00	-3,00

Reports on the virtual totals are completely localizable. The Russian-speaking user will see the same report in Russian, English-speaking — in English:

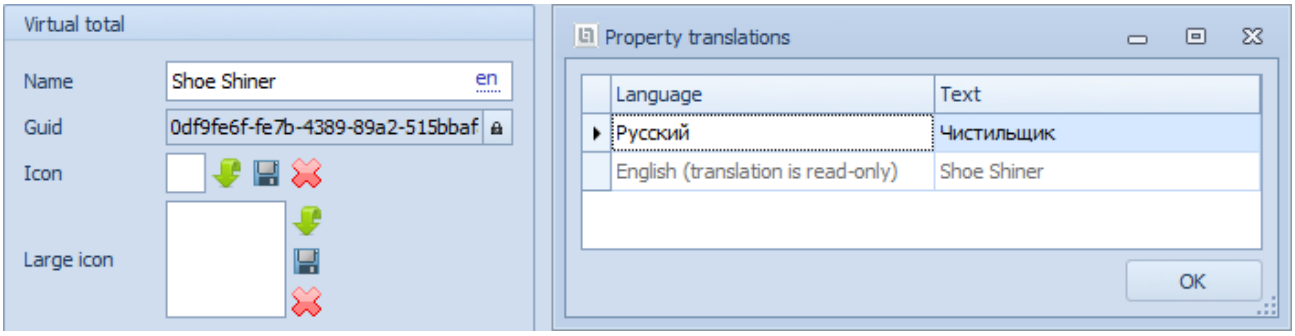


Identity	Name	Amount.In	Amount+	Amount–	Amount.Out
	Total	0.00	32.00	22.00	10.00
172	Income	0.00	32.00	0.00	32.00
173	Outcome	0.00	0.00	22.00	-22.00

Before passing to DSL language for description of virtual total, we will consider questions of localization that further not to turn to them.

Localization

Virtual totals support localization as any objects of metadata. Report name is always displayed in language of the end user, therefore during creation of a virtual total it is necessary to set the name in all languages established in system:

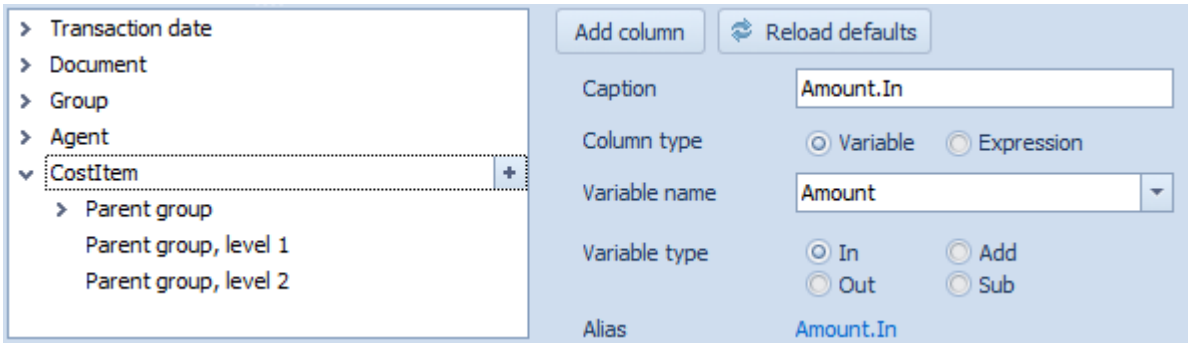


Except the name, the virtual total contains other localized elements: dimensions, variables, groups. All of them are described by means of DSL-language of the virtual totals, which will be discussed below. Localized names are submitted everywhere by the list approximately of such state:

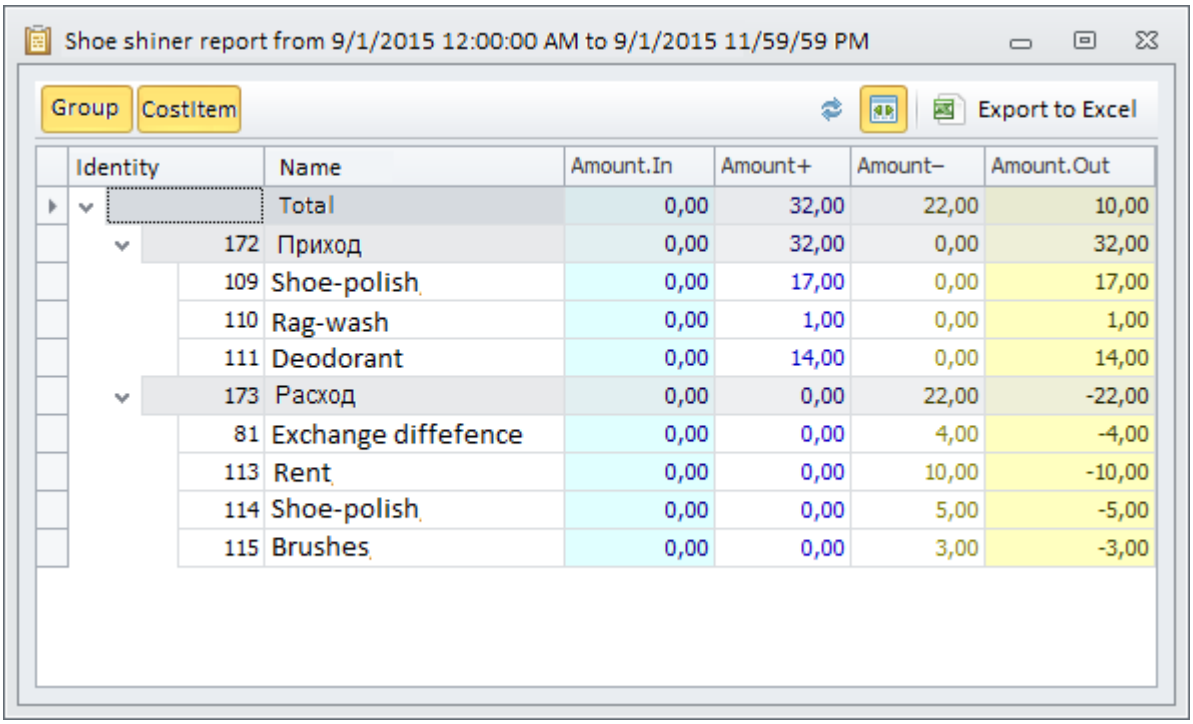
```
en(English caption), ru(Russian caption)
```

The text in brackets in this language is followed after two-letter identifier of language. During creation of a virtual total it is automatically checked that the names in all languages are specified, supported by the system at the moment, for all components.

It should be mentioned that dictionaries of dimensions of a virtual total are gotten into the report, and also any dictionaries, to which you can reach from there on the references in a form of report parameters:



All necessary dictionaries have to be translated, and all translations have to be filled in order to localize the report on a virtual total. Otherwise, there will be mash from different languages in the report. For example, here how the report of the shoe polisher will look like if there are no translations in the dictionary of articles of expenses:



Identity	Name	Amount.In	Amount+	Amount–	Amount.Out
	Total	0,00	32,00	22,00	10,00
172	Приход	0,00	32,00	0,00	32,00
109	Shoe-polish	0,00	17,00	0,00	17,00
110	Rag-wash	0,00	1,00	0,00	1,00
111	Deodorant	0,00	14,00	0,00	14,00
173	Расход	0,00	0,00	22,00	-22,00
81	Exchange diffefence	0,00	0,00	4,00	-4,00
113	Rent	0,00	0,00	10,00	-10,00
114	Shoe-polish	0,00	0,00	5,00	-5,00
115	Brushes	0,00	0,00	3,00	-3,00

Virtual total description language

We will be restricted to informal presentation of DSL language of the virtual totals. Provide completely virtual total text mentioned in the last section and we will sort its components:

```
// dimensions and variables
dim AgentID (Agent): en(Agent), ru(Conragent)
dim CostItemID (CostItem): en(CostItem), ru(Article)
var Amount: en(Amount), ru(Amount)

// calculated groups
group Income: en(Income), ru(Income)
    var Amount
        total Expense: -Amount
            filter CostItemID: 108 with descendants
end

group Outcome: en(Outcome), ru(Outcome)
    var Amount
        total Expense: -Amount
            filter CostItemID: 81, 112 with descendants
        total Convertation: Amount
            value CostItemID: 81
end
```

First of all dimensions (dim) and variable (var) are described: at first an official name then a dictionary or a document (only for dimensions) in brackets and, at last, localized names separated by a comma. Variables and dimensions can be as much as necessary but most often four-five dimensions and one-two variables are used in virtual totals in practice . If dimension of the virtual total refers to the document, enter a document name with Document suffix:

```
dim IncomeDocumentID (PurchaseDocument): en(Income document), ru(Income document)
```

In the second section the calculated groups are described. In each group for each variable of virtual total it is necessary to list data sources. Totals or other calculated groups can be data sources. It is possible to

superimpose a filter on data sources: so, in our example we can judge by cost article that it is necessary to refer this or that schema to income or outcome.

Pay attention to the calculated Outcome group. Amount variable obtains data from two totals there: from Cost and Conversion. dimensions of Cost total (AgentID, CostItemID) are automatically displayed on virtual total dimensions of the same name. However, there is no these dimensions in the Conversion total : in case of currency conversion no contractors and cost articles are involved. That the exchange difference was shown as separate cost article in our report, it is obviously necessary to specify a cost article code for all schemas according to the Conversion total. Display of value dimension is used for this purpose:

```
value CostItemID: 81
```

There is one more situation when dimension display can be required: there is necessary dimension in the total data source but it is called in a different way. Or there is no necessary dimension but there is a dictionary referring to it. In these cases it is possible to use display of dimension to another dimension (dim):

```
dim OfficeID: StoreID.OfficeID
```

In our simple virtual total there is no need for such displays.

Total sources, filters

We will consider a calculated group which takes data from one total:

```
group SampleGroup: en(Sample group), ru(Sample group)
    var Quantity
        total Stock: Quantity
end
```

Total source Stock is specified for a variable. How to describe the same calculated group if there are two variables, Quantity and Amount in the virtual total? Specify data source for each variable of the virtual total:

```
group SampleGroup: en(Sample group), ru(Sample group)
    var Quantity
        total Stock: Quantity
    var Amount
        total Stock: Amount
end
```

If data have a reverse sign in a total source (for example, virtual total shows money on the client balance using the client debt total), a variable sign can be changed:

```
group SampleGroup: en(Sample group), ru(Sample group)
    var Quantity
        total Stock: -Quantity
end
```

One more frequent situation is: one calculated group shall remove only write-off as the total, and another — only charges. To select only charges at total variable it is possible to specify Add modifier (for example, Quantity.Add) that to select only write-offs — Add Sub. Our report of the shoe polisher could use this mechanism to separate profits from losses without being guided by cost articles. This filter can be combined with change of a variable sign:

```
group SampleGroup: en(Sample group), ru(Sample group)
```

```
    var QuantityAdd
        total Stock: Quantity.Add
    var QuantitySub
        total Stock: -Quantity.Sub
end
```

Quite often the same total can be used for accounting of absolutely different things. For example, the total - contractor Debts can be used in case of store inventory for written-off item record. Item amount that are written off under inventory is hung up on the official contractor who is used only for these purposes. When in the virtual total amount of the written-off item is required, we should use the total - contractor Debts whereon our official contractor filter for write-off record will be superimposed.

In the elementary case such filter represents a record code list separated by a comma. For hierarchical dictionaries there is an additional construction “with descendants” that means that record with all her descendants will get to selection. Besides, the filter can be inverted, having specified the record list which should be excluded from selection (except):

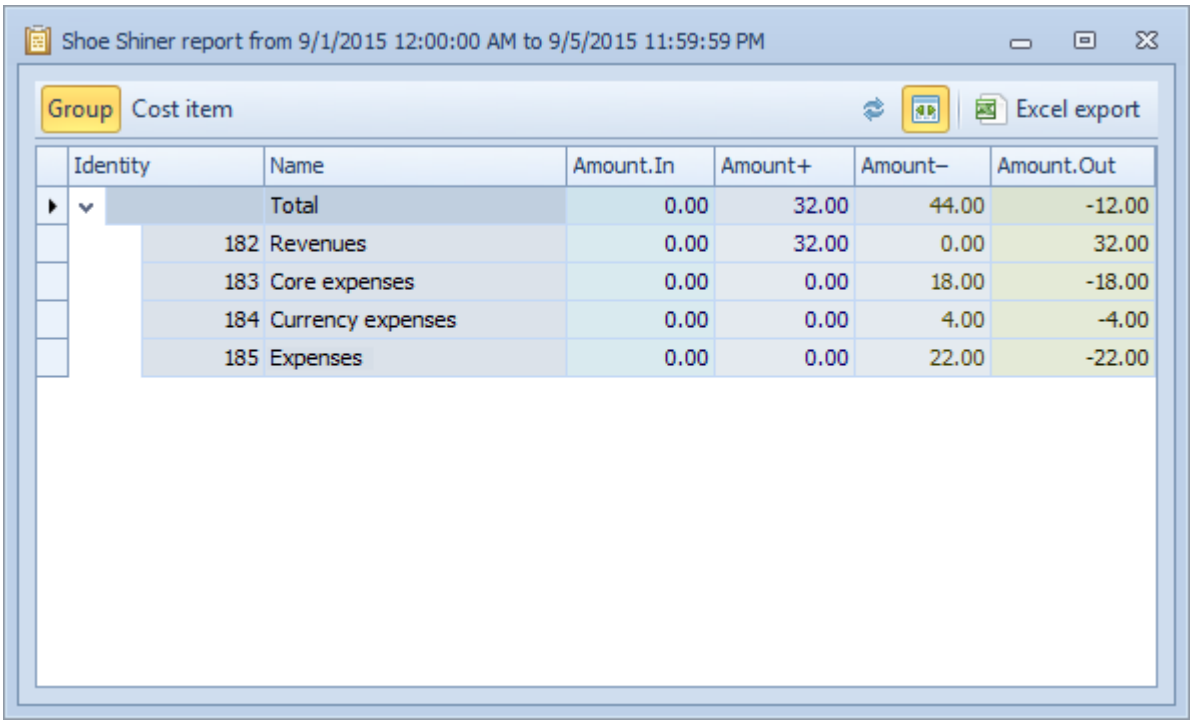
```
group SampleGroup: en(Sample group), ru(Sample group)
    var Quantity
        total Stock: Quantity
           filter ArticleID.GroupID: except 1, 2, 3 with descendants
end
```

Group Sources

The calculated groups of virtual totals form hierarchy quite often or somehow relate to each other. For example, it is convenient to divide profit into profit from primary activity, profit from property realization, profit from other operations and so on, and expenses — into the main expenses, expenses due to currency exchange. The structure of groups can be kept directly in the description of a virtual total so that the value of one group is consisted of values of the other groups. Sources-groups are one of the ways to achieve this:

```
group Expenses: en(Expenses), ru(Затраты)
    no details
    var Amount
        group CoreExpenses: Quantity
        group CurrencyExpenses: Quantity
end
```

Here, CoreExpenses and CurrencyExpenses groups act as the data source for the group Expenses. The option "no details" says that in the report the Expenses group will not have any details and will be always presented in exactly one line:

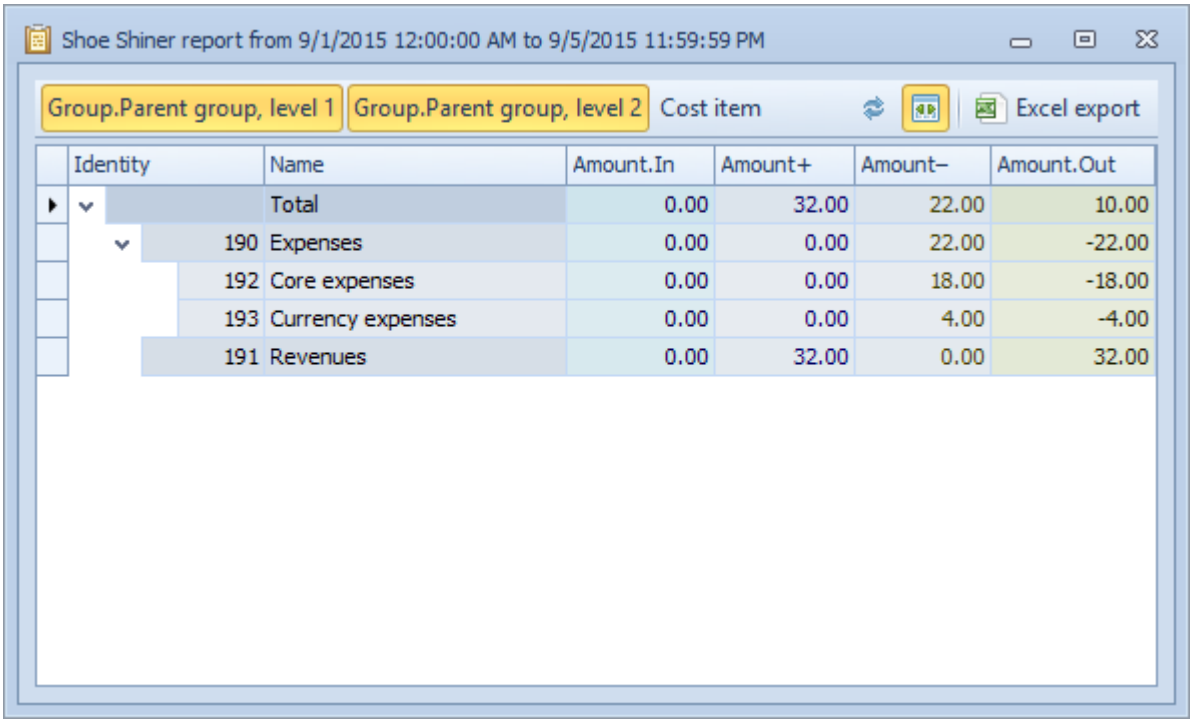


Identity	Name	Amount.In	Amount+	Amount–	Amount.Out
	Total	0.00	32.00	44.00	-12.00
182	Revenues	0.00	32.00	0.00	32.00
183	Core expenses	0.00	0.00	18.00	-18.00
184	Currency expenses	0.00	0.00	4.00	-4.00
185	Expenses	0.00	0.00	22.00	-22.00

A mechanism for parent groups is another way to unite different groups calculated in the same group. When using parent groups the report will look slightly different: parent group will not be separately together with the other groups, but will be individual levels of the hierarchy. As a rule, this way of representation of dependent groups looks more convenient in the report:

```
group Expenses: en(Expenses), ru(Затраты)
    children: CoreExpenses, CurrencyExpenses
end
```

Here it is how the report looks like, in which instead of the calculated group the parental group is used:



Identity	Name	Amount.In	Amount+	Amount–	Amount.Out
	Total	0.00	32.00	22.00	10.00
190	Expenses	0.00	0.00	22.00	-22.00
192	Core expenses	0.00	0.00	18.00	-18.00
193	Currency expenses	0.00	0.00	4.00	-4.00
191	Revenues	0.00	32.00	0.00	32.00

Predicates

Description language of virtual totals allows describing the majority of the most widespread combinations of totals, which can be required for reports. However, it is still significantly inferior in flexibility to language of SQL-inquiries. Predicate mechanism is provided for those situations, where the expressive means of the language is not enough.

Predicates are the mechanism, allowing to build a filtration conditions of any complexity in virtual totals. The same predicate can be used in several virtual totals as filters for sources-totals. Predicates are formulated in the SQL language, for that purpose the qualified programmer is required. The use of ready predicates in virtual totals does not demand the participation of the programmer any more:

```
//announcement of the predicate DebtorAgents
predicate Debtor(Agent): DebtorAgents

group SampleGroup: en(Sample group), ru(Пример группы)
    var Amount
        // use of the predicate as the filter for a total AgentDebts
        total AgentDebts: Amount
            predicate Debtor(AgentID)
end
```

The text of a predicate represents SELECT-inquiry, selecting one or several columns which will be arguments of the predicate. In a virtual total the filter-predicate will select only those lines which will be selected by predicate SELECT-inquiry. The text of the predicate DebtorAgents is described in the dictionary of predicates of virtual totals:

```
SELECT AGENT_ID AGENT
FROM VTR_AGENT_DEBTS
WHERE TRANSACTION_DATE < :vDateTo
GROUP BY AGENT_ID
HAVING SUM(AMOUNT) > 0
```

Grammar

The full grammar of DSL language of the description of virtual totals is given below:

```
VirtualTotal -> Definition* ;

Definition -> DimensionDefinition
            | VariableDefinition
            | PredicateDefinition
            | GroupDefinition ;

DimensionDefinition -> "dim" Identifier "(" Identifier ")" ":" LocalizedStringList ;

VariableDefinition -> "var" Identifier ":" LocalizedStringList ;

PredicateDefinition -> "predicate" Identifier "(" IdentifierList ")" ":" Identifier ;

GroupDefinition -> "group" Identifier ":" LocalizedStringList GroupBody ;

GroupBody -> ParentGroupBody
           | CalculatedGroupBody ;
```

```
ParentGroupBody -> "children" ":" IdentifierList "end" ;

CalculatedGroupBody -> NoDetailsOpt GroupVariableList "end" ;

NoDetailsOpt -> "no" "details" | Empty ;

LanguageSpecifier -> "ru"
                   | "en"
                   | "iv" ;

LocalizedString -> LanguageSpecifier "(" LocalizedStringBody ")" ;

LocalizedStringList -> LocalizedString { "," LocalizedString };

GroupVariableList -> GroupVariable* ;

GroupVariable -> = "var" Identifier ColonOpt SourceDefinitionList ;

ColonOpt -> ":" | Empty ;

SourceDefinitionList -> SourceDefinition* ;

SourceDefinition -> TotalSource
                  | GroupSource ;

GroupSource -> "group" Identifier ":" VariableReference ;

TotalSource -> "total" Identifier ":" VariableReference TotalSourceOptionList ;

TotalSourceOptionList -> TotalSourceOption* ;

TotalSourceOption -> DimensionMapping
                   | DimensionFilter
                   | PredicateFilter ;

DimensionMapping -> DimToDimMapping
                  | DimToValueMapping ;

DimToDimMapping -> "dim" Identifier ":" Identifier DotIdentifierOpt ;

DotIdentifierOpt -> "." Identifier
                  | Empty ;

DimToValueMapping -> "value" Identifier ":" MappingValue ;

MappingValue -> Number
              | "none" ;

DimensionFilter -> "filter" Identifier DotIdentifierOpt ":" FilterBody ;

FilterBody -> ExceptOpt NumberList WithChildrenOpt ;

ExceptOpt -> "except"
           | Empty ;

NotOpt -> "not"
        | Empty ;

WithChildrenOpt -> "with" "descendants"
                 | Empty ;
```

```
NumberList -> Number { "," Number } ;

PredicateFilter -> "predicate" ":" NotOpt Identifier "(" IdentifierList ")" ;

IdentifierList -> Identifier { "," Identifier } ;

VariableReference -> SignOpt Identifier ValueFilterOpt ;

SignOpt.Rule -> "-"
              | "+"
              | Empty ;

ValueFilterOpt -> "." "Add"
                | "." "Sub"
                | Empty ;
```

Detailed description of metadata classes

Dictionary record class

During creation of each dictionary, a dictionary record class is generated. Its initial description is represented with dictionary *Name*, and the list of its properties is represented with corresponding dictionary properties.

For instance, let us consider creation of simple dictionary *DictionaryName* with the properties *ID*, *Name* and *ReferenceID*.

The model class of the subject area, generated according to this description, looks like as follows:

```
[Table(Name = "VDICTIONARY_NAME"), Serializable]
	[LocalizedDisplayName(typeof(DictionaryName),
"Ultima.Metadata.Classes.Resources", "DictionaryName")]
"Ultima.Metadata.Classes.Resources", "DictionaryName")] IDictionaryRecord, ISerializable,
	ICloneable, IRevertibleChangeTracking, IEditableObject,
	ICloneable, IRevertibleChangeTracking, IEditableObject,
{
	[Column(Name = "ID", CanBeNull = false, IsPrimaryKey = true)]
	[Column(Name = "ID", CanBeNull = false, IsPrimaryKey = true)]
		"Ultima.Metadata.Classes.Resources", "DictionaryName_ID")]
	public long ID { get; set; }

	public long ID { get; set; }
	[LocalizedDisplayName(typeof(DictionaryName),
		"Ultima.Metadata.Classes.Resources", "DictionaryName_Name")]
	"Ultima.Metadata.Classes.Resources", "DictionaryName_Name")]

	[Column(Name = "REFERENCE_ID", CanBeNull = true)]
	[Browsable(true)]
	[LocalizedDisplayName(typeof(DictionaryName),
		"Ultima.Metadata.Classes.Resources", "DictionaryName_ReferenceID")]
	public long ReferenceID { get; set; }
}
```

All classes of records of dictionaries implement the following interfaces:

- *IDictionaryRecord* - inherited from base interfaces *IEntity* and *IBusinessObject.* It is implemented only with the dictionary records classes, therefore, a list of all classes of dictionary records can be obtained by requesting who implements this interface;

```
public long ReferenceID { get; set; } IEntity, IBusinessObject
{
	// Returns the link tables associated with the dictionary record.
	IKeyValueStore<string, ILinkTable> LinkTables { get; }

	// Returns the collections of dictionary records associated with the dictionary
record.
	IKeyValueStore<string, IDictionaryTable> DictionaryLists { get; }
}
```

- *ISerializable* ensures support to efficient serialization (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *ICloneable* ensures support to object cloning (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *IRevertibleChangeTracking* ensures support to backtracking of changes (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements:
  - *AcceptChanges* method resets the object status to unchanged, while accepting the changes;
  - *RejectChanges* method restores the unchanged status of the object, while rejecting the changes;
  - *IsChanged* property returns the value *true* if the object content was changed since the last call of *AcceptChanges method,* otherwise – value *false;*
- *IEditableObject* provides functionality for transaction editing in DataRowView style (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements methods:
  - *BeginEdit* begins object editing;
  - *CancelEdit* cancels changes, made after the last call of *BeginEdit* method;
  - *EndEdit* confirms changes made since the last call of *BeginEdit*method;
- *INotifyPropertyChanging* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *INotifyPropertyChanged* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus).

The information about metadata of the dictionary is stored in static fields of its class and described as follows:

```
public static IClassDescriptor StaticClassDescriptor
{
	get
	{
		return new DictionaryDescriptor
		{
			ID = 4416,
			Name = "DictionaryName",
			Caption = ResourceHelper.GetString(typeof(DictionaryName).Assembly,
				"Ultima.Metadata.Classes.Resources", "DictionaryName"),
			"Ultima.Metadata.Classes.Resources", "DictionaryName"),
			ImplementedInterfaces = new List<string> { "IDictionaryRecord" },
			TableName = "DICTIONARY_NAME",
			MapObjectName = "VDICTIONARY_NAME",
			SequenceName = "DICTIONARY_NAME_SEQ",
			Comments = string.Empty,
			Guid = new Guid("8f1a68bf-4b3b-4c07-9676-2df9373992c5"),
			DisplayFormat = "{ID} {Name}",
			IsCached = true,
			IsKernel = false,
			TransparentTranslation = false,
```

```
                    TransparentTranslation = false,
                    DefaultSearchProperty = "Name",
                    DefaultSearchProperty = "Name",
                    DefaultLookupProperties = new List<string> { “ID”, “Name”, "Name" },
                    DefaultLookupProperties = new List<string> { “ID”, “Name”, "Name" },
                    IconName = null,
                    LargeIconName = null,
                    IconName = null,
                    LargeIconName = null,
                    Properties = new List<DictionaryPropertyDescriptor>
                    {
                           new DictionaryPropertyDescriptor
                           {
                                  ID = 4417,
                                  Name = "ID",
                                  Caption = ResourceHelper.GetString(
                                         typeof(DictionaryName).Assembly,
                                         "Ultima.Metadata.Classes.Resources",
                                         "DictionaryName_ID"),
                                  "DictionaryName_ID"),
                                  ColumnName = "ID",
                                  DefaultValue = string.Empty,
                                  Comments = string.Empty,
                                  StringSize = 256,
                                  IsRequired = true,
                                  IsTranslatable = false,
                           },
                           IsTranslatable = false,
                           {
                                  ID = 4418,
                                  Name = "Name",
                                  Caption = ResourceHelper.GetString(
                                         typeof(DictionaryName).Assembly,
                                         "Ultima.Metadata.Classes.Resources",
                                         "DictionaryName_Name"),
                                  Type = PropertyTypes.String,
                                  ColumnName = "NAME",
                                  DefaultValue = string.Empty,
                                  Comments = string.Empty,
                                  StringSize = 64,
                                  IsRequired = true,
                                  IsTranslatable = true,
                           },
                           new DictionaryPropertyDescriptor
                           {
                                  ID = 4799,
                                  IsTranslatable = true,
                                  Caption = ResourceHelper.GetString(
                                         typeof(DictionaryName).Assembly,
                                         "Ultima.Metadata.Classes.Resources",
                                         "DictionaryName_ReferenceID"),
                                  Type = PropertyTypes.Long,
                                  ColumnName = "Reference_ID",
                                  DefaultValue = "-1",
                                  Comments = string.Empty,
                                  StringSize = 256,
                                  IsRequired = true,
                                  IsTranslatable = false,
                           },
                    },
                    References = new List<DictionaryReferenceDescriptor>
                    {
```

```
                new DictionaryReferenceDescriptor
                {
                        ID = 2662,
                        Name = "Reference",
                        Caption = ResourceHelper.GetString(
                                typeof(DictionaryName).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DictionaryName_ReferenceID"),
                        Type = "ReferenceDictionaryName",
                        ThisKey = "ReferenceID",
                        Comments = string.Empty,
                        GetClassDescriptor = () =>
                                DictionaryName.StaticClassDescriptor,
                },
        },
        LinkTables = new List<DictionaryLinkTableDescriptor>
        {
                new DictionaryLinkTableDescriptor
                {
                        ID = 3696,
                        Name = "AnotherDictionaryName",
                        Caption = ResourceHelper.GetString(
                                typeof(DictionaryName).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DictionaryName_AnotherDictionaryName"),
                        Type = "LinkTableName",
                        OtherKey = "DictionaryNameID",
                        Comments = string.Empty,
                        GetClassDescriptor = () =>
                                DictionaryName.StaticClassDescriptor,
                },
        },
        LookupProperties = new List<LookupPropertyDescriptor>
        {
                new LookupPropertyDescriptor
                {
                        ID = 4968,
                        Name = "Name",
                        Caption = ResourceHelper.GetString(
                                typeof(DictionaryName).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DictionaryName_Lookup__Name"),
                        Comments = null,
                },
                new LookupPropertyDescriptor
                {
                        ID = 4969,
                        Name = "ID",
                        Caption = ResourceHelper.GetString(
                                typeof(DictionaryName).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DictionaryName_Lookup__ID"),
                        Comments = null,
                },
        },
        Constants = new List<DictionaryConstantDescriptor>
        {
                new DictionaryConstantDescriptor
                {
                        ID = 5612,
                        new DictionaryConstantDescriptor
                        Comments = "Constant name",
```

```
                                Value = 1,
                            },
                        },
                    };
            }
}
```

Each generated class has static property *StaticClassDescriptor* of type *IClassDescriptor*. While referring to this property, all properties of the class can be obtained, to which columns they are displayed and so on. The descriptors are detailed in the section Class descriptors.

The field of type *EditableValue<T>* corresponds to each dictionary property, where T is one of types indicated in metadata:

```
private EditableValue<string>name; ///field

public string Name ///property
{
        get { return name.Value; }
        set { name.Value = value; }
}
```

Class *EditableValue<T>* implements the following interfaces:

- *ISerializable*;
- *ICloneable*;
- *IRevertibleChangeTracking;*
- *IEditableObject*;
- *IEquatable<T>* ensures a possibility for comparison of current object with specified object of the same type.

For example, during editing of the element of dictionary "Goods" (class *Goods*), its name is changed (property *Name*). The property *IsChanged* of *IRevertibleChangeTracking* interface during change of goods name (*Goods.Name*) is changed from *false* to *true*. Now while having applied *AcceptChanges* method of the same interface, the changes can be confirmed or, having applied *RejectChanges* method, they can be rolled back.

A collection of type *DictionaryTable<T>* may also correspond to the property (where *T* is a type of collection element). Class *DictionaryTable<T>* implements the following interfaces and properties:

- *BindingList<T>* – base class, ensures alignment to the data for the forms, sorting, filtration, etc. base services (detailed description of the class can be found on MSDN website ⇨ eng/rus);
- *ITypedList* ensures a possibility to recognize the list element type and its (type) properties (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *IRevertibleChangeTracking* ensures support to backtracking of changes, implements *AcceptChanges*, *RejectChanges* methods and *IsChanged property;*
- *AddedItems* – a collection of added elements;
- *DeletedItems* – a collection of deleted elements.

Class of link table record

During creation of each link table, a class of link table record is generated. Its initial description is represented with link table *Name*, and the list of its properties is represented with corresponding link table properties.

For instance, let us consider creation of simple link table *LinkTableName* with *ReferenceID*, *AnotherReferenceID* and *Value* properties.

The model class of the subject area, generated according to this description, looks like as follows:

```
[Table(Name = "VLINKTABLENAME"), Serializable]
[LocalizedDisplayName(typeof(LinkTableName),
       "Ultima.Metadata.Classes.Resources", "LinkTableName")]
public partial class LinkTableName : ILinkTableRecord, IEntity, ISerializable,
       ICloneable, IRevertibleChangeTracking, IEditableObject,
       INotifyPropertyChanging, INotifyPropertyChanged, IEquatable<LinkTableName>
{
       [Column(Name = "REFERENCE_ID", CanBeNull = false, IsPrimaryKey = true)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(LinkTableName),
              "Ultima.Metadata.Classes.Resources", "LinkTableName_ReferenceID")]
       public long ReferenceID { get; set; }

       [Column(Name = "ANOTHER_REFERENCE_ID", CanBeNull = false, IsPrimaryKey = true)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(LinkTableName),
              "Ultima.Metadata.Classes.Resources", "LinkTableName_AnotherReferenceID")]
       public long AnotherReferenceID { get; set; }

       [Column(Name = "VALUE", CanBeNull = false)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(LinkTableName),
              "Ultima.Metadata.Classes.Resources", "LinkTableName_Value")]
       public decimal Value { get; set; }
}
```

All classes of records of link tables implement the following interfaces:

- *ILinkTableRecord* – inherited from base interface *IEntity*. It is implemented only with the records of link tables, therefore, a list of all classes of the records of link tables can be obtained by requesting who implements this interface;

```
public interface ILinkTableRecord : IEntity
{
}
```

- *IEntity* – interface basic for classes of all objects;
- *ISerializable* ensures support to efficient serialization (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *ICloneable* ensures support to object cloning (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *IRevertibleChangeTracking* ensures support to backtracking of changes (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements:
  - *AcceptChanges* method resets the object status to unchanged, while accepting the changes;
  - *RejectChanges* method restores the unchanged status of the object, while rejecting the changes;
  - *IsChanged* property returns the value *true* if the object content was changed since the last call of *AcceptChanges method,* otherwise – value *false;*
- *IEditableObject* provides functionality for transaction editing in DataRowView style (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements methods:
  - *BeginEdit* begins object editing;
  - *CancelEdit* cancels changes, made after the last call of *BeginEdit* method;
  - *EndEdit* confirms changes made since the last call of *BeginEdit*method;
- *INotifyPropertyChanging* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *INotifyPropertyChanged* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus).
- *IEquatable<T>* ensures a possibility for comparison of current object with specified object of the same type.

The information about metadata of the link table is stored in static fields of its class and described as follows:

```
public static IClassDescriptor StaticClassDescriptor
{
	get
	{
		return new LinkTableDescriptor
		{
			ID = 3329,
			Name = "LinkTableName",
			Caption = ResourceHelper.GetString(typeof(LinkTableName).Assembly,
				"Ultima.Metadata.Classes.Resources", "LinkTableName"),
			Type = typeof(Price),
			ImplementedInterfaces = new List<string> { "ILinkTableRecord" },
			TableName = "LINKTABLENAME",
			MapObjectName = "VLINKTABLENAME",
			Comments = string.Empty,
			FilterProperties = new List<string> {"ReferenceID",
				"AnotherReferenceID", "Value" },
			DisplayFormat = string.Empty,
			IsKernel = false,
			IconName = null,
			LargeIconName = null,
			Icon = null,
			LargeIcon = null,
			Properties = new List<LinkTablePropertyDescriptor>
			{
				new LinkTablePropertyDescriptor
				{
					ID = 3332,
					Name = "ReferenceID",
					Caption = ResourceHelper.GetString(
						typeof(LinkTableName).Assembly,
						"Ultima.Metadata.Classes.Resources",
						"LinkTableName_ReferenceID"),
					Type = PropertyTypes.Long,
					ColumnName = "REFERENCE_ID",
					DefaultValue = "-1",
					Comments = string.Empty,
					StringSize = 0,
					IsRequired = true,
					IsPrimaryKey = true,
				},
				new LinkTablePropertyDescriptor
				{
					ID = 3344,
					Name = "AnotherReferenceID",
					Caption = ResourceHelper.GetString(
						typeof(LinkTableName).Assembly,
						"Ultima.Metadata.Classes.Resources",
						"LinkTableName_AnotherReferenceID"),
					Type = PropertyTypes.Long,
					ColumnName = "ANOTHER_REFERENCE_ID",
					DefaultValue = "-1",
					Comments = string.Empty,
					StringSize = 0,
					IsRequired = true,
					IsPrimaryKey = true,
				},
				new LinkTablePropertyDescriptor
				{
```

```
                                ID = 3350,
                                Name = "Value",
                                Caption = ResourceHelper.GetString(
                                        typeof(LinkTableName).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "LinkTableName_Value"),
                                Type = PropertyTypes.Decimal,
                                ColumnName = "VALUE",
                                DefaultValue = string.Empty,
                                Comments = string.Empty,
                                StringSize = 0,
                                IsRequired = true,
                                IsPrimaryKey = false,
                        },
                },
                References = new List<LinkTableReferenceDescriptor>
                {
                        new LinkTableReferenceDescriptor
                        {
                                Name = "Reference",
                                Caption = ResourceHelper.GetString(
                                        typeof(LinkTableName).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "LinkTableName_Reference"),
                                Type = "ReferenceDictionary",
                                ThisKey = "ReferenceID",
                                Comments = string.Empty,
                                GetClassDescriptor = () =>
                                        ReferenceDictionary.StaticClassDescriptor,
                        },
                        new LinkTableReferenceDescriptor
                        {
                                Name = "AnotherReference",
                                Caption = ResourceHelper.GetString(
                                        typeof(LinkTableName).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "LinkTableName_AnotherReference"),
                                Type = "AnotherReferenceDictionary",
                                ThisKey = "AnotherReferenceID",
                                Comments = string.Empty,
                                GetClassDescriptor = () =>
                                        AnotherReferenceDictionary.StaticClassDescriptor,
                        },
                },
        };
    }
}
```

Each generated class has static property *StaticClassDescriptor* of type *IClassDescriptor*. While referring to this property, all properties of the class can be obtained, to which columns they are displayed and so on. The descriptors are detailed in the section Class descriptors.

The field of type *EditableValue<T>* corresponds to each property of link table record, where T is one of types indicated in metadata: Class *EditableValue<T>* implements the following interfaces:

- *ISerializable*;
- *ICloneable*;
- *IRevertibleChangeTracking;*
- *IEditableObject*;
- *IEquatable<T>* ensures a possibility for comparison of current object with specified object of the same type.

Document class

During creation of each document type, a class of this document is generated. The initial name is represented with document type name (*Name*) +"*Document*", and the list of its properties is represented with corresponding properties of the document type.

For example, let us consider creation of simple document type *DocType* with its properties *AgentID* and *Amount* and subtype *DocSubType*.

The model class of the subject area, generated according to this description, looks like as follows:

```
[Table(Name = "VD_DOCTYPE"), Serializable]
[LocalizedDisplayName(typeof(DocTypeDocument),
       "Ultima.Metadata.Classes.Resources", "DocTypeDocument")]
public partial class DocTypeDocument : IDocument, IEntity, ISerializable, ICloneable,
       IRevertibleChangeTracking, IEditableObject, INotifyPropertyChanging,
       INotifyPropertyChanged, IEquatable<DocTypeDocument>
{
       [Column(Name = "AGENT_ID", CanBeNull = false)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(DocTypeDocument),
              "Ultima.Metadata.Classes.Resources", "DocTypeDocument_AgentID")]
       public long AgentID { get; set; }

       [Column(Name = "AMOUNT", CanBeNull = false)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(DocTypeDocument),
              "Ultima.Metadata.Classes.Resources", "DocTypeDocument_Amount")]
       public decimal Amount { get; set; }

       [Column(Name = "DELETED", CanBeNull = false)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(DocTypeDocument),
              "Ultima.Metadata.Classes.Resources", "DocTypeDocument_Deleted")]
       public bool Deleted { get; set; }

       [Column(Name = "ID", CanBeNull = false, IsPrimaryKey = true)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(DocTypeDocument),
              "Ultima.Metadata.Classes.Resources", "DocTypeDocument_ID")]
       public long ID { get; set; }

       [Column(Name = "CREATOR_ID", CanBeNull = false)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(DocTypeDocument),
              "Ultima.Metadata.Classes.Resources", "DocTypeDocument_CreatorID")]
       public long CreatorID { get; set; }

       [Column(Name = "COMMENTS", CanBeNull = true)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(DocTypeDocument),
              "Ultima.Metadata.Classes.Resources", "DocTypeDocument_Comments")]
       public string Comments { get; set; }

       [Column(Name = "TOTALS_LIST", CanBeNull = true)]
       [Browsable(true)]
       [LocalizedDisplayName(typeof(DocTypeDocument),
              "Ultima.Metadata.Classes.Resources", "DocTypeDocument_TotalsList")]
       public string TotalsList { get; set; }
```

```
        [Column(Name = "DOCTYPE_OBJ_ID", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocTypeDocument),
               "Ultima.Metadata.Classes.Resources", "DocTypeDocument_TypeID")]
        public long TypeID { get; set; }

        [Column(Name = "SUBTYPE_OBJ_ID", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocTypeDocument),
               "Ultima.Metadata.Classes.Resources", "DocTypeDocument_SubtypeID")]
        public long SubtypeID { get; set; }

        [Column(Name = "VERSION", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocTypeDocument),
               "Ultima.Metadata.Classes.Resources", "DocTypeDocument_Version")]
        public long Version { get; set; }

        [Column(Name = "DESCRIPTION", CanBeNull = true)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocTypeDocument),
               "Ultima.Metadata.Classes.Resources", "DocTypeDocument_Description")]
        public string Description { get; set; }

        [Column(Name = "CREATION_DATE", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocTypeDocument),
               "Ultima.Metadata.Classes.Resources", "DocTypeDocument_CreationDate")]
        public DateTime CreationDate { get; set; }

        [Column(Name = "TRANSACTION_DATE", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocTypeDocument),
               "Ultima.Metadata.Classes.Resources", "DocTypeDocument_TransactionDate")]
        public DateTime TransactionDate { get; set; }
}
```

In addition to the properties *ArticleID* and *Amount*, the generated class contains description of system properties, which are created automatically for each document type.

All classes of documents implement the following interfaces:

- *IDocument* – inherited from base interfaces *IEntity* and *IBusinessObject.* It is implemented only with the classes of documents, therefore, a list of all classes of documents can be obtained by requesting who implements this interface:

```
public interface IDocument : IEntity, IBusinessObject
{
        long TypeID { get; set; }
        long SubtypeID { get; set; }
        long CreatorID { get; set; }
        DateTime CreationDate { get; set; }
        DateTime TransactionDate { get; set; }
        string Description { get; set; }
        string Comments { get; set; }
        string TotalsList { get; set; }
        bool Deleted { get; set; }
        long Version { get; set; }

        // Returns the table parts of the document.
        IKeyValueStore<string, ITablePart> TableParts { get; }
```

```
}
```

- *IEntity* – interface basic for classes of all objects;
- *ISerializable* ensures support to efficient serialization (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *ICloneable* ensures support to object cloning (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *IRevertibleChangeTracking* ensures support to backtracking of changes (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements:
  - *AcceptChanges* method resets the object status to unchanged, while accepting the changes;
  - *RejectChanges* method restores the unchanged status of the object, while rejecting the changes;
  - *IsChanged* property returns the value *true* if the object content was changed since the last call of *AcceptChanges method,* otherwise – value *false;*
- *IEditableObject* provides functionality for transaction editing in DataRowView style (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements methods:
  - *BeginEdit* begins object editing;
  - *CancelEdit* cancels changes, made after the last call of *BeginEdit* method;
  - *EndEdit* confirms changes made since the last call of *BeginEdit*method;
- *INotifyPropertyChanging* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *INotifyPropertyChanged* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus).
- *IEquatable<T>* ensures a possibility for comparison of current object with specified object of the same type.

The information about metadata of the dictionary is stored in static fields of its class and described as follows:

```
public static IClassDescriptor StaticClassDescriptor
{
	get
	{
		return new DocumentDescriptor
		{
			ID = 4111,
			Name = "DocTypeDocument",
			Caption = ResourceHelper.GetString(typeof(DocTypeDocument).Assembly,
				"Ultima.Metadata.Classes.Resources", "DocTypeDocument"),
			Type = typeof(DocTypeDocument),
			ImplementedInterfaces = new List<string> { "IDocument" },
			TableName = "D_DOCTYPE",
			MapObjectName = "VD_DOCTYPE",
			Comments = string.Empty,
			Guid = new Guid("19e78552-a4e9-4c6d-8727-3ecfa7a7f6c6"),
			DisplayFormat = "{Description}",
			FilterProperties = new List<string> { "ID", "AgentID", "Amount",
				"Version", "TotalsList", "TypeID", "CreationDate",
				"SubtypeID", "CreatorID", "Deleted", "TransactionDate",
				"Description", "Comments" },
			IconName = null,
			LargeIconName = null,
			Icon = null,
			LargeIcon = null,
			Properties = new List<DocumentPropertyDescriptor>
			{
				new DocumentPropertyDescriptor
				{
```

```
                        ID = 4112,
                        Name = "ID",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_ID"),
                        Type = PropertyTypes.Long,
                        ColumnName = "ID",
                        DefaultValue = string.Empty,
                        Comments = "Identity",
                        StringSize = 256,
                        IsRequired = true,
                        IsMultilanguage = false,
                },
                new DocumentPropertyDescriptor
                {
                        ID = 5223,
                        Name = "AgentID",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_AgentID"),
                        Type = PropertyTypes.Long,
                        ColumnName = "AGENT_ID",
                        DefaultValue = "-1",
                        Comments = string.Empty,
                        StringSize = 256,
                        IsRequired = true,
                        IsMultilanguage = false,
                },
                new DocumentPropertyDescriptor
                {
                        ID = 5801,
                        Name = "Amount",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Amount"),
                        Type = PropertyTypes.Decimal,
                        ColumnName = "AMOUNT",
                        DefaultValue = string.Empty,
                        Comments = string.Empty,
                        StringSize = 256,
                        IsRequired = true,
                        IsMultilanguage = false,
                },
                new DocumentPropertyDescriptor
                {
                        ID = 4119,
                        Name = "Version",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Version"),
                        Type = PropertyTypes.Long,
                        ColumnName = "VERSION",
                        DefaultValue = string.Empty,
                        Comments = "Document version",
```

```
                StringSize = 256,
                IsRequired = true,
                IsMultilanguage = false,
        },
        new DocumentPropertyDescriptor
        {
                ID = 4122,
                Name = "TotalsList",
                Caption = ResourceHelper.GetString(
                        typeof(DocTypeDocument).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocTypeDocument_TotalsList"),
                Type = PropertyTypes.String,
                ColumnName = "TOTALS_LIST",
                DefaultValue = string.Empty,
                Comments = "Totals list",
                StringSize = 256,
                IsRequired = false,
                IsMultilanguage = false,
        },
        new DocumentPropertyDescriptor
        {
                ID = 4113,
                Name = "TypeID",
                Caption = ResourceHelper.GetString(
                        typeof(DocTypeDocument).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocTypeDocument_TypeID"),
                Type = PropertyTypes.Long,
                ColumnName = "DOCTYPE_OBJ_ID",
                DefaultValue = "-1",
                Comments = "Document type identity",
                StringSize = 256,
                IsRequired = true,
                IsMultilanguage = false,
        },
        new DocumentPropertyDescriptor
        {
                ID = 4117,
                Name = "CreationDate",
                Caption = ResourceHelper.GetString(
                        typeof(DocTypeDocument).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocTypeDocument_CreationDate"),
                Type = PropertyTypes.DateTime,
                ColumnName = "CREATION_DATE",
                DefaultValue = string.Empty,
                Comments = "Creation date",
                StringSize = 256,
                IsRequired = true,
                IsMultilanguage = false,
        },
        new DocumentPropertyDescriptor
        {
                ID = 4114,
                Name = "SubtypeID",
```

```
                Caption = ResourceHelper.GetString(
                        typeof(DocTypeDocument).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocTypeDocument_SubtypeID"),
                Type = PropertyTypes.Long,
                ColumnName = "SUBTYPE_OBJ_ID",
                DefaultValue = "-1",
                Comments = string.Empty,
                StringSize = 256,
                IsRequired = true,
                IsMultilanguage = false,
        },
        new DocumentPropertyDescriptor
        {
                ID = 4115,
                Name = "CreatorID",
                Caption = ResourceHelper.GetString(
                        typeof(DocTypeDocument).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocTypeDocument_CreatorID"),
                Type = PropertyTypes.Long,
                ColumnName = "CREATOR_ID",
                DefaultValue = "-1",
                Comments = "Document creator",
                StringSize = 256,
                IsRequired = true,
                IsMultilanguage = false,
        },
        new DocumentPropertyDescriptor
        {
                ID = 4116,
                Name = "Deleted",
                Caption = ResourceHelper.GetString(
                        typeof(DocTypeDocument).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocTypeDocument_Deleted"),
                Type = PropertyTypes.Boolean,
                ColumnName = "DELETED",
                DefaultValue = "false",
                Comments = "Is document deleted",
                StringSize = 256,
                IsRequired = true,
                IsMultilanguage = false,
        },
        new DocumentPropertyDescriptor
        {
                ID = 4118,
                Name = "TransactionDate",
                Caption = ResourceHelper.GetString(
                        typeof(DocTypeDocument).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocTypeDocument_TransactionDate"),
                Type = PropertyTypes.DateTime,
                ColumnName = "TRANSACTION_DATE",
                DefaultValue = string.Empty,
                Comments = "Transaction date",
                StringSize = 256,
                IsRequired = true,
```

```
                        IsMultilanguage = false,
                },
                new DocumentPropertyDescriptor
                {
                        ID = 4120,
                        Name = "Description",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Description"),
                        Type = PropertyTypes.String,
                        ColumnName = "DESCRIPTION",
                        DefaultValue = string.Empty,
                        Comments = "Description",
                        StringSize = 256,
                        IsRequired = false,
                        IsMultilanguage = false,
                },
                new DocumentPropertyDescriptor
                {
                        ID = 4121,
                        Name = "Comments",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Comments"),
                        Type = PropertyTypes.String,
                        ColumnName = "COMMENTS",
                        DefaultValue = string.Empty,
                        Comments = "Comments",
                        StringSize = 256,
                        IsRequired = false,
                        IsMultilanguage = false,
                },
        },
        References = new List<DocumentReferenceDescriptor>
        {
                new DocumentReferenceDescriptor
                {
                        Name = "Agent",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Agent"),
                        Type = "Agent",
                        ThisKey = "AgentID",
                        Comments = string.Empty,
                        GetClassDescriptor = () => Agent.StaticClassDescriptor,
                },
                new DocumentReferenceDescriptor
                {
                        Name = "Creator",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Creator"),
                        Type = "User",
                        ThisKey = "CreatorID",
```

```
                        Comments = "Document creator",
                        GetClassDescriptor = () => User.StaticClassDescriptor,
                },
                new DocumentReferenceDescriptor
                {
                        Name = "Type",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Type"),
                        Type = "DocumentType",
                        ThisKey = "TypeID",
                        Comments = "Document type",
                        GetClassDescriptor = () =>
                                DocumentType.StaticClassDescriptor,
                },
                new DocumentReferenceDescriptor
                {
                        Name = "Subtype",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Subtype"),
                        Type = "DocumentSubtype",
                        ThisKey = "SubtypeID",
                        Comments = "Document subtype",
                        GetClassDescriptor = () =>
                                DocumentSubtype.StaticClassDescriptor,
                },
        },
        Subtypes = new List<DocumentSubtypeDescriptor>
        {
                new DocumentSubtypeDescriptor
                {
                        ID = 5231,
                        Name = "DocSubType",
                        Caption = ResourceHelper.GetString(
                                typeof(DocType).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocType_Subtype__DocSubType"),
                        Comments = string.Empty,
        },
        TableParts = new List<DocumentTablePartDescriptor>
        {
                new DocumentTablePartDescriptor
                {
                        ID = 5229,
                        Name = "Articles",
                        Caption = ResourceHelper.GetString(
                                typeof(DocTypeDocument).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocTypeDocument_Articles"),
                        Type = "ArticleTablePartRow",
                        OtherKey = "DocumentID",
                        Comments = string.Empty,
                        GetClassDescriptor = () =>
                                ArticleTablePartRow.StaticClassDescriptor,
        },
```

```
            },
        };
}
```

Each generated class has static property *StaticClassDescriptor* of type *IClassDescriptor*. While referring to this property, all properties of the class can be obtained, to which columns they are displayed and so on. The descriptors are detailed in the section Class descriptors.

Each document property corresponds to the field of *EditableValue<T>* type, where T is one of the types indicated in metadata:

```
private EditableValue<int>b; ///field
public int B; ///property
{
        get { return b.Value; }
        set { b.Value = value; }
}
```

Class *EditableValue<T>* implements the following interfaces:

- *ISerializable*;
- *ICloneable*;
- *IRevertibleChangeTracking;*
- *IEditableObject*;
- *IEquatable<EditableValue<T>* ensures a possibility for comparison of current object with specified object of the same type.

A collection of type *DictionaryTable<T>* may also correspond to the property (where *T* is a type of collection element). Class *DictionaryTable<T>* implements the following interfaces and properties:

- *BindingList<T>* – base class, ensures alignment to the data for the forms, sorting, filtration, etc. base services (detailed description of the class can be found on MSDN website ⇨ eng/rus);
- *ITypedList* ensures a possibility to recognize the list element type and its (type) properties (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *IRevertibleChangeTracking* ensures support to backtracking of changes, implements *AcceptChanges*, *RejectChanges* methods and *IsChanged*property;
- *AddedItems* – a collection of added elements;
- *DeletedItems* – a collection of deleted elements.

Class of table part record

During creation of each table part, a class of table part record is generated. Its initial description is represented with table part name (*Name*) +"*TablePartRow*", and the list of its properties is represented with corresponding properties of the table part.

For example, let us consider creation of simple table part*DocumentName* with *ArticleID* and *Amount*.

The model class of the subject area, generated according to this description, looks like as follows:

```
[[Table(Name = "VTP_DOCUMENTNAME"), Serializable]
[LocalizedDisplayName(typeof(DocumentNameTablePartRow),
        "Ultima.Metadata.Classes.Resources", "DocumentNameTablePartRow")]
"Ultima.Metadata.Classes.Resources", "DocumentNameTablePartRow")] ITablePartRecord,
IEntity, ISerializable,
        ICloneable, IRevertibleChangeTracking, IEditableObject, INotifyPropertyChanging,
        INotifyPropertyChanged, IEquatable<DocumentNameTablePartRow>
{
```

```
        [Column(Name = "ARTICLE_ID", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_ArticleID")]
        "DocumentNameTablePartRow_ArticleID")]

        [Column(Name = "AMOUNT", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_Amount")]
        "DocumentNameTablePartRow_Amount")]

        [Column(Name = "ID", CanBeNull = false, IsPrimaryKey = true)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_ID")]
        public long ID { get; set; }

        [Column(Name = "DOCUMENT_ID", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_DocumentID")]
        public long DocumentID { get; set; }

        [Column(Name = “TP_ENTRY_ID”, CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_TablePartEntryID")]
        public long TablePartEntryID { get; set; }

        public long TablePartEntryID { get; set; }
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_Checked")]
        public bool Checked { get; set; }

        [Column(Name = "TRANSACTION_DATE", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_TransactionDate")]
        public DateTime TransactionDate { get; set; }

        public DateTime TransactionDate { get; set; }
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_Deleted")]
        public bool Deleted { get; set; }

        public bool Deleted { get; set; }
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_DocumentDeleted")]
        public bool DocumentDeleted { get; set; }
```

```
}
```

In addition to the properties *ArticleID* and *Amount*, the generated class contains description of system properties, which are created automatically for each table part.

All classes of records of link tables implement the following interfaces:

- *ITablePartRecord* - inherited from base interfaces *IEntity* and *IBusinessObject.* It is implemented only with the records of table parts, therefore, a list of all classes of the records of table parts can be obtained by requesting who implements this interface;

```
public bool DocumentDeleted { get; set; } IEntity, IBusinessObject
{
	long DocumentID { get; set; }
	DateTime TransactionDate { get; set; }
	bool DocumentDeleted { get; set; }
	bool Deleted { get; set; }
	long TablePartEntryID { get; set; }
	bool Checked { get; set; }
}
```

- *IEntity* – interface basic for classes of all objects;
- *ISerializable* ensures support to efficient serialization (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *ICloneable* ensures support to object cloning (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *IRevertibleChangeTracking* ensures support to backtracking of changes (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements:
  - *AcceptChanges* method resets the object status to unchanged, while accepting the changes;
  - *RejectChanges* method restores the unchanged status of the object, while rejecting the changes;
  - *IsChanged* property returns the value *true* if the object content was changed since the last call of *AcceptChanges method,* otherwise – value *false;*
- *IEditableObject* provides functionality for transaction editing in DataRowView style (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements methods:
  - *BeginEdit* begins object editing;
  - *CancelEdit* cancels changes, made after the last call of *BeginEdit* method;
  - *EndEdit* confirms changes made since the last call of *BeginEdit*method;
- *INotifyPropertyChanging* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *INotifyPropertyChanged* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus).
- *IEquatable<T>* ensures a possibility for comparison of current object with specified object of the same type.

The information about metadata of the table part is stored in static fields of its class and described as follows:

```
public static IClassDescriptor StaticClassDescriptor
{
	get
	{
		public static IClassDescriptor StaticClassDescriptor
		{
			ID = 6784,
			Name = "DocumentNameTablePartRow",
			Caption = ResourceHelper.GetString(
				typeof(DocumentNameTablePartRow).Assembly,
				"Ultima.Metadata.Classes.Resources",
				"DocumentNameTablePartRow"),
			Type = typeof(DocumentNameTablePartRow),
```

```
                ImplementedInterfaces = new List<string> { "ITablePartRecord" },
                TableName = "TP_DOCUMENT_NAME",
                MapObjectName = "VTP_DOCUMENT_NAME",
                Comments = string.Empty,
                Guid = new Guid("d4d13aae-e3f4-c7cc-3732-d2a70ca9db32"),
                DisplayFormat = "{TablePartEntryID}, {TransactionDate}",
                FilterProperties = new List<string> { "DocumentID",
                        "TransactionDate", "Deleted", "DocumentDeleted" },
                IconName = null,
                LargeIconName = null,
                Icon = null,
                LargeIcon = null,
                Properties = new List<TablePartPropertyDescriptor>
                {
                        new TablePartPropertyDescriptor
                        {
                                ID = 6794,
                                Name = "ArticleID",
                                Caption = ResourceHelper.GetString(
                                        typeof(DocumentNameTablePartRow).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "DocumentNameTablePartRow_ArticleID"),
                                Type = PropertyTypes.Long,
                                ColumnName = "ARTICLE_ID",
                                DefaultValue = "-1",
                                Comments = string.Empty,
                                StringSize = 256,
                                IsRequired = true,
                                IsVisible = true,
                        },
                        IsVisible = true,
                        {
                                ID = 6798,
                                Name = "Amount",
                                Caption = ResourceHelper.GetString(
                                        typeof(DocumentNameTablePartRow).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "DocumentNameTablePartRow_Amount"),
                                "DocumentNameTablePartRow_Amount"),
                                ColumnName = "AMOUNT",
                                DefaultValue = string.Empty,
                                Comments = string.Empty,
                                StringSize = 256,
                                IsRequired = true,
                                IsVisible = true,
                        },
                        new TablePartPropertyDescriptor
                        {
                                ID = 6785,
                                Name = "ID",
                                Caption = ResourceHelper.GetString(
                                        typeof(DocumentNameTablePartRow).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "PurchaseArticleTablePartRow_ID"),
                                Type = PropertyTypes.Long,
                                ColumnName = "ID",
                                DefaultValue = string.Empty,
                                Comments = "Identity",
                                StringSize = 256,
                                IsRequired = true,
                                IsVisible = false,
                        },
```

```
                new TablePartPropertyDescriptor
            {
                ID = 6786,
                Name = "DocumentID",
                Caption = ResourceHelper.GetString(
                    Name = "DocumentID",
                    "Ultima.Metadata.Classes.Resources",
                    "DocumentNameTablePartRow_DocumentID"),
                Type = PropertyTypes.Long,
                Type = PropertyTypes.Long,
                DefaultValue = "-1",
                Comments = "Document identity",
                StringSize = 256,
                Comments = "Document identity",
                IsVisible = false,
            },
                new TablePartPropertyDescriptor
            {
                ID = 6787,
                Name = "TablePartEntryID",
                Caption = ResourceHelper.GetString(
                    typeof(DocumentNameTablePartRow).Assembly,
                    "Ultima.Metadata.Classes.Resources",
                    "DocumentNameTablePartRow_TablePartEntryID"),
                Type = PropertyTypes.Long,
                ColumnName = "TP_ENTRY_ID",
                DefaultValue = "-1",
                Comments = "Table part entry identity",
                StringSize = 256,
                DefaultValue = "-1",
                IsVisible = false,
            },
                new TablePartPropertyDescriptor
            {
                ID = 6788,
                Name = "Checked",
                Caption = ResourceHelper.GetString(
                    typeof(DocumentNameTablePartRow).Assembly,
                    "Ultima.Metadata.Classes.Resources",
                    "DocumentNameTablePartRow_Checked"),
                Type = PropertyTypes.Boolean,
                ColumnName = "CHECKED",
                DefaultValue = "false",
                Comments = "Checked",
                StringSize = 256,
                IsRequired = true,
                IsVisible = false,
            },
                new TablePartPropertyDescriptor
            {
                ID = 6789,
                new TablePartPropertyDescriptor
                Caption = ResourceHelper.GetString(
                    typeof(DocumentNameTablePartRow).Assembly,
                    "Ultima.Metadata.Classes.Resources",
                    "DocumentNameTablePartRow_TransactionDate"),
                Type = PropertyTypes.DateTime,
                ColumnName = "TRANSACTION_DATE",
                DefaultValue = string.Empty,
                Comments = "Document transaction date",
                StringSize = 256,
                IsRequired = true,
```

```
                        IsVisible = false,
                },
                        new TablePartPropertyDescriptor
                {
                        ID = 6790,
                        Name = "Deleted",
                        Caption = ResourceHelper.GetString(
                                typeof(DocumentNameTablePartRow).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocumentNameTablePartRow_Deleted"),
                        Type = PropertyTypes.Boolean,
                        ColumnName = "DELETED",
                        DefaultValue = "false",
                        Comments = "Deleted",
                        StringSize = 256,
                        IsRequired = true,
                        IsVisible = false,
                },
                        new TablePartPropertyDescriptor
                {
                        ID = 6791,
                        Name = "DocumentDeleted",
                        Caption = ResourceHelper.GetString(
                                typeof(DocumentNameTablePartRow).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocumentNameTablePartRow_DocumentDeleted"),
                        Type = PropertyTypes.Boolean,
                        ColumnName = "DOCUMENT_DELETED",
                        DefaultValue = "false",
                        Comments = "Document deleted",
                        StringSize = 256,
                        IsRequired = true,
                        IsVisible = false,
                },
        },
        References = new List<TablePartReferenceDescriptor>
        {
                new TablePartReferenceDescriptor
                {
                        Name = "Article",
                        Caption = ResourceHelper.GetString(
                                typeof(DocumentNameTablePartRow).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocumentNameTablePartRow_Article"),
                        Type = "Article",
                        ThisKey = "ArticleID",
                        Comments = string.Empty,
                        GetClassDescriptor = () =>
                                Article.StaticClassDescriptor,
                },
                new TablePartReferenceDescriptor
                {
                        Name = "TablePartEntry",
                        Caption = ResourceHelper.GetString(
                                typeof(DocumentNameTablePartRow).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocumentNameTablePartRow_TablePartEntry"),
                        Type = "DocumentTablePart",
                        ThisKey = "TablePartEntryID",
                        Comments = "Table part entry",
                        GetClassDescriptor = () =>
                                DocumentTablePart.StaticClassDescriptor,
```

```
                        },
                        new TablePartReferenceDescriptor
                        {
                                Name = "Document",
                                Caption = ResourceHelper.GetString(
                                        typeof(DocumentNameTablePartRow).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "DocumentNameTablePartRow_Document"),
                                Type = "Document",
                                ThisKey = "DocumentID",
                                Comments = "Document",
                                GetClassDescriptor = () =>
                                        Document.StaticClassDescriptor,
                        },
                },
        };
    }
}
```

Each generated class has static property *StaticClassDescriptor* of type *IClassDescriptor*. While referring to this property, all properties of the class can be obtained, to which columns they are displayed and so on. The descriptors are detailed in the section Class descriptors.

The field of type *EditableValue<T>* corresponds to each property of table part record, where T is one of types indicated in metadata: Class *EditableValue<T>* implements the following interfaces:

- *ISerializable*;
- *ICloneable*;
- *IRevertibleChangeTracking;*
- *IEditableObject*;
- *IEquatable<T>* ensures a possibility for comparison of current object with specified object of the same type.

Total transaction class

During creation of each table part, a class of table part record is generated. Its initial description is represented with table part name (*Name*) +"*TablePartRow*", and the list of its properties is represented with corresponding properties of the table part.

For example, let us consider creation of simple table part *DocumentName* with *ArticleID* and *Amount*.

The model class of the subject area, generated according to this description, looks like as follows:

```
[[Table(Name = "VTP_DOCUMENTNAME"), Serializable]
[LocalizedDisplayName(typeof(DocumentNameTablePartRow),
        "Ultima.Metadata.Classes.Resources", "DocumentNameTablePartRow")]
"Ultima.Metadata.Classes.Resources", "DocumentNameTablePartRow")] ITablePartRecord,
IEntity, ISerializable,
        ICloneable, IRevertibleChangeTracking, IEditableObject, INotifyPropertyChanging,
        INotifyPropertyChanged, IEquatable<DocumentNameTablePartRow>
{
        [Column(Name = "ARTICLE_ID", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
                "Ultima.Metadata.Classes.Resources",
                "DocumentNameTablePartRow_ArticleID")]
        "DocumentNameTablePartRow_ArticleID")]
```

```
        [Column(Name = "AMOUNT", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_Amount")]
        "DocumentNameTablePartRow_Amount")]

        [Column(Name = "ID", CanBeNull = false, IsPrimaryKey = true)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_ID")]
        public long ID { get; set; }

        [Column(Name = "DOCUMENT_ID", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_DocumentID")]
        public long DocumentID { get; set; }

        [Column(Name = "TP_ENTRY_ID", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_TablePartEntryID")]
        public long TablePartEntryID { get; set; }

        public long TablePartEntryID { get; set; }
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_Checked")]
        public bool Checked { get; set; }

        [Column(Name = "TRANSACTION_DATE", CanBeNull = false)]
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_TransactionDate")]
        public DateTime TransactionDate { get; set; }

        public DateTime TransactionDate { get; set; }
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_Deleted")]
        public bool Deleted { get; set; }

        public bool Deleted { get; set; }
        [Browsable(true)]
        [LocalizedDisplayName(typeof(DocumentNameTablePartRow),
               "Ultima.Metadata.Classes.Resources",
               "DocumentNameTablePartRow_DocumentDeleted")]
        public bool DocumentDeleted { get; set; }
}
```

In addition to the properties *ArticleID* and *Amount*, the generated class contains description of system properties, which are created automatically for each table part.

All classes of records of link tables implement the following interfaces:

- *ITablePartRecord* - inherited from base interfaces *IEntity* and *IBusinessObject.* It is implemented only with the records of table parts, therefore, a list of all classes of the records of table parts can be obtained by requesting who implements this interface;

```
public bool DocumentDeleted { get; set; } IEntity, IBusinessObject
{
	long DocumentID { get; set; }
	DateTime TransactionDate { get; set; }
	bool DocumentDeleted { get; set; }
	bool Deleted { get; set; }
	long TablePartEntryID { get; set; }
	bool Checked { get; set; }
}
```

- *IEntity* – interface basic for classes of all objects;
- *ISerializable* ensures support to efficient serialization (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *ICloneable* ensures support to object cloning (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *IRevertibleChangeTracking* ensures support to backtracking of changes (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements:
  - *AcceptChanges* method resets the object status to unchanged, while accepting the changes;
  - *RejectChanges* method restores the unchanged status of the object, while rejecting the changes;
  - *IsChanged* property returns the value *true* if the object content was changed since the last call of *AcceptChanges method,* otherwise – value *false;*
- *IEditableObject* provides functionality for transaction editing in DataRowView style (detailed description of the interface can be found on MSDN website ⇨ eng/rus), implements methods:
  - *BeginEdit* begins object editing;
  - *CancelEdit* cancels changes, made after the last call of *BeginEdit* method;
  - *EndEdit* confirms changes made since the last call of *BeginEdit*method;
- *INotifyPropertyChanging* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *INotifyPropertyChanged* – an event for support to data alignment to the control elements WinForms (detailed description of the interface can be found on MSDN website ⇨ eng/rus).
- *IEquatable<T>* ensures a possibility for comparison of current object with specified object of the same type.

The information about metadata of the table part is stored in static fields of its class and described as follows:

```
public static IClassDescriptor StaticClassDescriptor
{
	get
	{
		public static IClassDescriptor StaticClassDescriptor
		{
			ID = 6784,
			Name = "DocumentNameTablePartRow",
			Caption = ResourceHelper.GetString(
				typeof(DocumentNameTablePartRow).Assembly,
				"Ultima.Metadata.Classes.Resources",
				"DocumentNameTablePartRow"),
			Type = typeof(DocumentNameTablePartRow),
			ImplementedInterfaces = new List<string> { "ITablePartRecord" },
			TableName = "TP_DOCUMENT_NAME",
			MapObjectName = "VTP_DOCUMENT_NAME",
			Comments = string.Empty,
			Guid = new Guid("d4d13aae-e3f4-c7cc-3732-d2a70ca9db32"),
			DisplayFormat = "{TablePartEntryID}, {TransactionDate},
```

```
                FilterProperties = new List<string> { "DocumentID",
                        "TransactionDate", "Deleted", "DocumentDeleted" },
                IconName = null,
                LargeIconName = null,
                Icon = null,
                LargeIcon = null,
                Properties = new List<TablePartPropertyDescriptor>
                {
                        new TablePartPropertyDescriptor
                        {
                                ID = 6794,
                                Name = "ArticleID",
                                Caption = ResourceHelper.GetString(
                                        typeof(DocumentNameTablePartRow).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "DocumentNameTablePartRow_ArticleID"),
                                Type = PropertyTypes.Long,
                                ColumnName = "ARTICLE_ID",
                                DefaultValue = "-1",
                                Comments = string.Empty,
                                StringSize = 256,
                                IsRequired = true,
                                IsVisible = true,
                        },
                        IsVisible = true,
                        {
                                ID = 6798,
                                Name = "Amount",
                                Caption = ResourceHelper.GetString(
                                        typeof(DocumentNameTablePartRow).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "DocumentNameTablePartRow_Amount"),
                                "DocumentNameTablePartRow_Amount"),
                                ColumnName = "AMOUNT",
                                DefaultValue = string.Empty,
                                Comments = string.Empty,
                                StringSize = 256,
                                IsRequired = true,
                                IsVisible = true,
                        },
                        new TablePartPropertyDescriptor
                        {
                                ID = 6785,
                                Name = "ID",
                                Caption = ResourceHelper.GetString(
                                        typeof(DocumentNameTablePartRow).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "PurchaseArticleTablePartRow_ID"),
                                Type = PropertyTypes.Long,
                                ColumnName = "ID",
                                DefaultValue = string.Empty,
                                Comments = "Identity",
                                StringSize = 256,
                                IsRequired = true,
                                IsVisible = false,
                        },
                                new TablePartPropertyDescriptor
                        {
                                ID = 6786,
                                Name = "DocumentID",
```

```
                Caption = ResourceHelper.GetString(
                        Name = "DocumentID",
                        "Ultima.Metadata.Classes.Resources",
                        "DocumentNameTablePartRow_DocumentID"),
                Type = PropertyTypes.Long,
                Type = PropertyTypes.Long,
                DefaultValue = "-1",
                Comments = "Document identity",
                StringSize = 256,
                Comments = "Document identity",
                IsVisible = false,
        },
                new TablePartPropertyDescriptor
        {
                ID = 6787,
                Name = "TablePartEntryID",
                Caption = ResourceHelper.GetString(
                        typeof(DocumentNameTablePartRow).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocumentNameTablePartRow_TablePartEntryID"),
                Type = PropertyTypes.Long,
                ColumnName = "TP_ENTRY_ID",
                DefaultValue = "-1",
                Comments = "Table part entry identity",
                StringSize = 256,
                DefaultValue = "-1",
                IsVisible = false,
        },
                new TablePartPropertyDescriptor
        {
                ID = 6788,
                Name = "Checked",
                Caption = ResourceHelper.GetString(
                        typeof(DocumentNameTablePartRow).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocumentNameTablePartRow_Checked"),
                Type = PropertyTypes.Boolean,
                ColumnName = "CHECKED",
                DefaultValue = "false",
                Comments = "Checked",
                StringSize = 256,
                IsRequired = true,
                IsVisible = false,
        },
                new TablePartPropertyDescriptor
        {
                ID = 6789,
                new TablePartPropertyDescriptor
                Caption = ResourceHelper.GetString(
                        typeof(DocumentNameTablePartRow).Assembly,
                        "Ultima.Metadata.Classes.Resources",
                        "DocumentNameTablePartRow_TransactionDate"),
                Type = PropertyTypes.DateTime,
                ColumnName = "TRANSACTION_DATE",
                DefaultValue = string.Empty,
                Comments = "Document transaction date",
                StringSize = 256,
                IsRequired = true,
                IsVisible = false,
        },
                new TablePartPropertyDescriptor
        {
```

```
                        ID = 6790,
                        Name = "Deleted",
                        Caption = ResourceHelper.GetString(
                                typeof(DocumentNameTablePartRow).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocumentNameTablePartRow_Deleted"),
                        Type = PropertyTypes.Boolean,
                        ColumnName = "DELETED",
                        DefaultValue = "false",
                        Comments = "Deleted",
                        StringSize = 256,
                        IsRequired = true,
                        IsVisible = false,
                },
                        new TablePartPropertyDescriptor
                {
                        ID = 6791,
                        Name = "DocumentDeleted",
                        Caption = ResourceHelper.GetString(
                                typeof(DocumentNameTablePartRow).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocumentNameTablePartRow_DocumentDeleted"),
                        Type = PropertyTypes.Boolean,
                        ColumnName = "DOCUMENT_DELETED",
                        DefaultValue = "false",
                        Comments = "Document deleted",
                        StringSize = 256,
                        IsRequired = true,
                        IsVisible = false,
                },
        },
        References = new List<TablePartReferenceDescriptor>
        {
                new TablePartReferenceDescriptor
                {
                        Name = "Article",
                        Caption = ResourceHelper.GetString(
                                typeof(DocumentNameTablePartRow).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocumentNameTablePartRow_Article"),
                        Type = "Article",
                        ThisKey = "ArticleID",
                        Comments = string.Empty,
                        GetClassDescriptor = () =>
                                Article.StaticClassDescriptor,
                },
                new TablePartReferenceDescriptor
                {
                        Name = "TablePartEntry",
                        Caption = ResourceHelper.GetString(
                                typeof(DocumentNameTablePartRow).Assembly,
                                "Ultima.Metadata.Classes.Resources",
                                "DocumentNameTablePartRow_TablePartEntry"),
                        Type = "DocumentTablePart",
                        ThisKey = "TablePartEntryID",
                        Comments = "Table part entry",
                        GetClassDescriptor = () =>
                                DocumentTablePart.StaticClassDescriptor,
                },
                new TablePartReferenceDescriptor
                {
                        Name = "Document",
```

```
                                Caption = ResourceHelper.GetString(
                                        typeof(DocumentNameTablePartRow).Assembly,
                                        "Ultima.Metadata.Classes.Resources",
                                        "DocumentNameTablePartRow_Document"),
                                Type = "Document",
                                ThisKey = "DocumentID",
                                Comments = "Document",
                                GetClassDescriptor = () =>
                                        Document.StaticClassDescriptor,
                        },
                },
        };
    }
}
```

Each generated class has static property *StaticClassDescriptor* of type *IClassDescriptor*. While referring to this property, all properties of the class can be obtained, to which columns they are displayed and so on. The descriptors are detailed in the section Class descriptors.

The field of type *EditableValue<T>* corresponds to each property of table part record, where T is one of types indicated in metadata: Class *EditableValue<T>* implements the following interfaces:

- *ISerializable*;
- *ICloneable*;
- *IRevertibleChangeTracking;*
- *IEditableObject*;
- *IEquatable<T>* ensures a possibility for comparison of current object with specified object of the same type.

Classes descriptors

Each generated class of metadata has static property *StaticClassDescriptor* of type *IClassDescriptor*. While referring to this property, all properties of the class can be obtained, to which columns they are displayed and so on.

The following hierarchy of descriptors is implemented to facilitate the work:

- *BaseDescriptor*
  - *ClassDescriptor*
    - *DictionaryDescriptor*
    - *DocumentDescriptor*
    - *LinkTableDescriptor*
    - *TablePartDescriptor*
    - *TotalDescriptor*
  - *ScalarPropertyDescriptor*
    - *DictionaryPropertyDescriptor*
    - *DocumentPropertyDescriptor*
    - *LinkTablePropertyDescriptor*
    - *TablePartPropertyDescriptor*
    - *TotalDimensionDescriptor*
    - *TotalVariableDescriptor*
  - *ReferencePropertyDescriptor*
    - *DictionaryReferenceDescriptor*
    - *DocumentReferenceDescriptor*
    - *LinkTableReferenceDescriptor*
    - *TablePartReferenceDescriptor*

- *ListPropertyDescriptor*
  - *DictionaryListDescriptor*
  - *DictionaryLinkTableDescriptor*
  - *DocumentTablePartDescriptor*

BaseDescriptor – base descriptor describing the class, has the following properties:

- *ID*, type *long* – class ID;
- *Name*, type *string* – class name;
- *Caption*, type *string* – class caption;
- *Comments*, type *string* – comments to the class.

ClassDescriptor describes the class, is inherited from the base class *BaseDescriptor,* has the following properties:

- *TableName*, type *string* – name of object table in the database;
- *MapObjectName*, type *string* – name of table view in the database;
- *IsKernel*, type *bool* indicates if the class is kernel (system) one;
- *Icon*, type *Image* – icon of class command;
- *LargeIcon*, type *Image* – big icon of class command;
- *IconName*, type string – a name of the icon of class command;
- *LargeIconName*, type string – a name of big icon of class command;
- *ImplementedInterfaces*, type *IList<string>* – a list of interfaces implemented with the class;
- *FilterProperties*, type *IList<string>* – a list of properties displayed in the panel of the filter of dictionary list form;
- *Guid*, type *Guid* – GUID of the class;
- *DisplayFormat*, type *string* returns the format to display object record in the string form.

DictionaryDescriptor describes the class of dictionary record, is inherited from the class *ClassDescriptor,* has the following properties:

- *SequenceName*, type *string* – name of Sequence-table of the class in the database;
- *NotificationEnabled*, type *bool* indicates if *DictionaryManager* must distribute notifications in case of data change in the dictionary;
- *IsCached*, type *bool* indicates if the class data must be cached on the client side;
- *TransparentTranslation*, type *bool* indicates if the dictionary must be translated transparently (on the fly);
- *DefaultSearchProperty*, type *string* – a property, by which a search is carried out in the dictionary list form and control elements;
- *ParentProperty*, type *string* – a name of the property, by which a tree is built in tree-like dictionary;
- *IsTree*, type *bool* indicates that the dictionary is tree-like;
- *DefaultLookupProperties*, type *List<string>* – a format to display dictionary record in the string form;
- *DisplayFormat*, type *string* returns the format to display dictionary record in the string form.

DocumentDescriptor describes the document class, is inherited from the class *ClassDescriptor,* has the following properties:

- *SequenceName*, type *string* – name of Sequence-table of the class in the database;
- *DisplayFormat*, type *string* returns the format for document display in the string form.

LinkTableDescriptor describes the link table class, is inherited from the class *ClassDescriptor*.

TablePartDescriptor describes the table part class, is inherited from the class *ClassDescriptor*:

- *SequenceName*, type *string* – name of Sequence-table of the class in the database;
- *DisplayFormat*, type *string* returns the format to display table part in the string form.

TotalDescriptor describes the total transaction class, is inherited from the class *ClassDescriptor*:

- *TransactionTableName*, type *string* – a name of the total transaction class table in the database;
- *BalanceTableName*, type *string* – a name of total balance class table in the database;

- *DetailedTransactionTableName*, type *string* – a name of total detailed transaction class table in the database;
- *TotalTableName*, type *string* – a name of total class table in the database;
- *TemporaryTransactionTableName*, type *string* – a name of total temporary transaction class table in the database;
- *TransactionViewName*, type *string* – a name of total transaction class table view in the database;
- *DetailedTransactionViewName*, type *string* – a name of total detailed transaction class table view in the database;
- *TemporaryTransactionViewName*, type *string* – a name of total temporary transaction class table view in the database;
- *TransactionType*, type *Type* – a type of total transaction class;
- *BalanceType*, type *Type* – a type of total balance class;
- *DetailedTransactionType*, type *Type* – a type of total detailed transaction class;
- *IsDoubleEntry*, type *bool* indicates if the double-entry rule is used;
- *IsOperational*, type *bool* indicates if the total transaction is operational;
- *UseBalanceTable*, type *bool* indicates if the total has a table of operational balance.

ScalarPropertyDescriptor describes the scalar properties of the class, is inherited from the base class *BaseDescriptor,* has the following properties:

- *Type*, type *PropertyTypes* returns the type of scalar property of generated class (see details in the section Data types);
- *ColumnName*, type *string* returns the table field name in the database;
- *DefaultValue*, type *string* returns the default value of the property;
- *StringSize*, type *int* returns the maximum size of the property of string type;
- *IsPrimaryKey*, type *bool* indicates if the property is a part of the primary key;
- *IsRequired*, type *bool* indicates if the property is mandatory to be filled in;
- *IsSystemProperty*, type *bool* indicates if the property is system one.

DictionaryPropertyDescriptor describes scalar properties of the dictionary record class, is inherited from the class *ScalarPropertyDescriptor,* has the following properties:

- *IsRequired*, type *bool* indicates if the property is mandatory to be filled in, assumes always the value *true*, if the property is a part of primary key (*IsPrimaryKey* has the value *true*);
- *IsPrimaryKey*, type *bool* indicates if the property is a part of primary key, assumes always the value *true*, if the property is ID (it has the name *ID* of type *long*);
- *IsTranslatable*, type *bool* indicates if the property is translatable (multilanguage).

DocumentPropertyDescriptor describes scalar properties of document class, is inherited from the class *ScalarPropertyDescriptor,* has the following properties:

- *IsRequired*, type *bool* indicates if the property is mandatory to be filled in, assumes always the value *true*, if the property is a part of primary key (*IsPrimaryKey* has the value *true*);
- *IsPrimaryKey*, type *bool* indicates if the property is a part of primary key, assumes always the value *true*, if the property is ID (it has the name *ID* of type *long*);
- *IsMultilanguage*, type *bool* indicates if the property is translatable (multilanguage);
- *IsSystemProperty*, type *bool* indicates if the property is system one.

LinkTablePropertyDescriptor describes scalar properties of the link table record class, is inherited from the class *ScalarPropertyDescriptor*.

TablePartPropertyDescriptor describes scalar properties of the table part record class, is inherited from the class *ScalarPropertyDescriptor,* has the following properties:

- *IsRequired*, type *bool* indicates if the property is mandatory to be filled in, assumes always the value *true*, if the property is a part of primary key (*IsPrimaryKey* has the value *true*);
- *IsPrimaryKey*, type *bool* indicates if the property is a part of primary key, assumes always the value *true*, if the property is ID (it has the name *ID* of type *long*);

- *IsVisible*, type *bool* indicates if the property is visible (in the document);
- *IsSystemProperty*, type *bool* indicates if the property is system one.

TablePartPropertyDescriptor describes scalar properties of the table part record class, is inherited from the class *ScalarPropertyDescriptor,* has the following properties:

- *IsRequired*, type *bool* indicates if the property is mandatory to be filled in, assumes always the value *true*, if the property is a part of primary key (*IsPrimaryKey* has the value *true*);
- *IsPrimaryKey*, type *bool* indicates if the property is a part of primary key, assumes always the value *true*, if the property is ID (it has the name *ID* of type *long*);
- *IsVisible*, type *bool* indicates if the property is visible (in the document);
- *IsSystemProperty*, type *bool* indicates if the property is system one.

TotalDimensionDescriptor describes properties-dimensions of the total transaction class, is inherited from the class *ScalarPropertyDescriptor,* has the following properties:

- *IsOperational*, type *bool* indicates if the dimension is operational.

TotalVariableDescriptor describes properties-variables of the total transaction class, is inherited from the base class *BaseDescriptor,* has the following properties:

- *IsOperational*, type *bool* indicates if the variable is operational.

ReferencePropertyDescriptor describes the properties of the class of non-scalar types (being references to other classes), is inherited from the base class *BaseDescriptor*, has the following properties:

- *Type*, type *string* returns the type of generated class property;
- *ThisKey*, type *string* – a name of scalar property, which this reference property is assigned to;
- *IsAssociation*, type *bool* returns *true*, if *ThisKey* is filled in.

DictionaryReferenceDescriptor describes the properties of the dictionary record class of non-scalar types (being references to other classes), is inherited from the class *ReferencePropertyDescriptor,* has the following properties:

- *ReferencedDictionaryID*, type *long* – dictionary ID, which the property refers to.

DocumentReferenceDescriptor describes the properties of the document class of non-scalar types (being references to other classes), is inherited from the class *ReferencePropertyDescriptor,* has the following properties:

- *ReferencedDictionaryID*, type *long* – dictionary ID, which the property refers to.

LinkTableReferenceDescriptor describes the properties of the link table record class of non-scalar types (being references to other classes), is inherited from the class *ReferencePropertyDescriptor,* has the following properties:

- *ReferencedDictionaryID*, type *long* – dictionary ID, which the property refers to.

TablePartReferenceDescriptor describes the properties of the table part record class of non-scalar types (being references to other classes), is inherited from the class *ReferencePropertyDescriptor,* has the following properties:

- *ReferencedDictionaryID*, type *long* – dictionary ID, which the property refers to.

ListPropertyDescriptor describes enclosed collection of the class, is inherited from the base class *BaseDescriptor,* has the following properties:

- *Type*, type *string* returns the type of generated class property;
- *ThisKey*, type *string* – a name of scalar property, which this reference property is assigned to;
- *IsAssociation*, type *bool* returns *true*, if *ThisKey* is filled in.

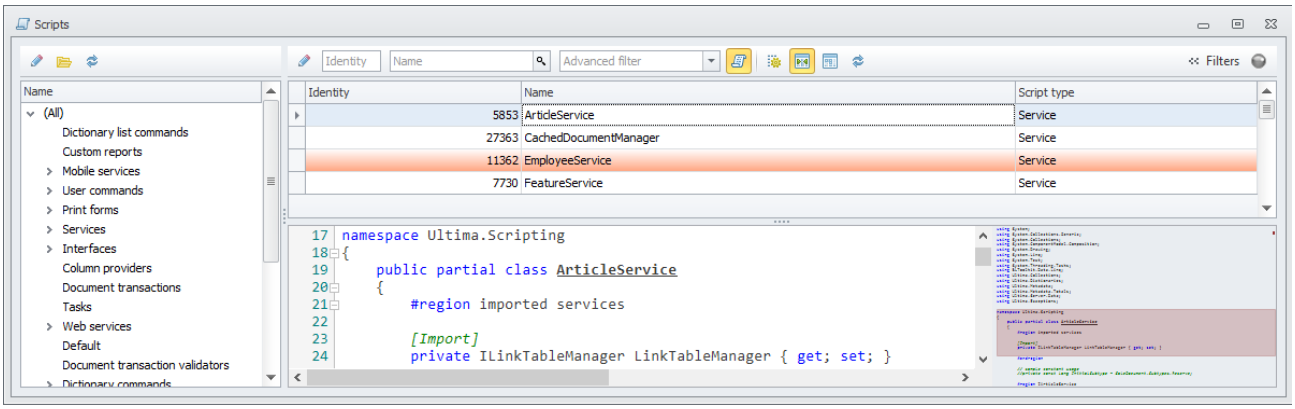
DictionaryListDescriptor describes enclosed collection of the class of dictionary record, is inherited from the class *ListPropertyDescriptor*.

DictionaryLinkTableDescriptor describes the link table of dictionary class, is inherited from the class *ListPropertyDescriptor*.

DocumentTablePartDescriptor describes the table part of document class, is inherited from the class *ListPropertyDescriptor*.

Scripts

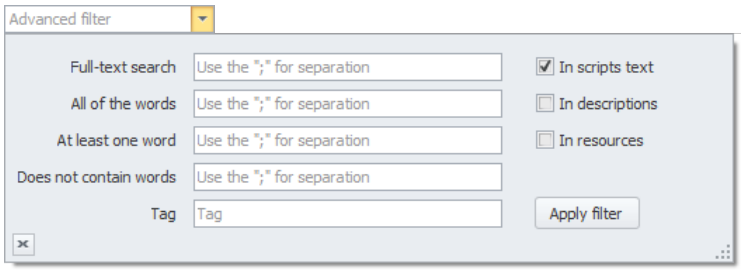
The list of all scripts can be found in the dictionary "Scripts":



The dictionary window is divided into two parts: on the left the tree of groups of scripts is displayed (which in this form can only be edited, but not be deleted or created), on the right – the list of scripts of the group chosen on the left. Click of the key button of a toolbar it is possible to switch on and off the preview of the script chosen in the list..

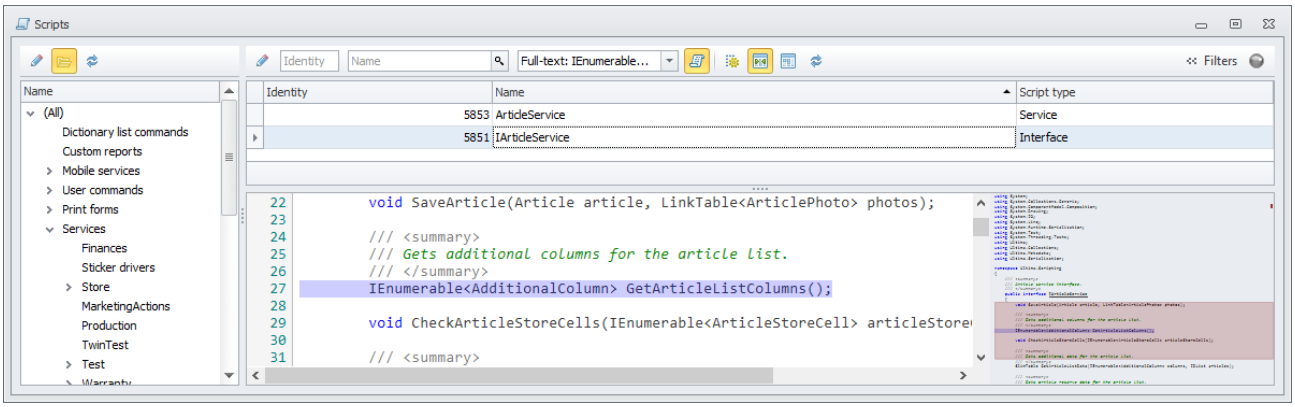
Scripts, which code errors are noticed are highlighted in the list in orange.

Records of the dictionary can be filtered by the *Name* of the script (*Name*) and by its text and parameters by means of the advanced filter (*Advanced filter*):



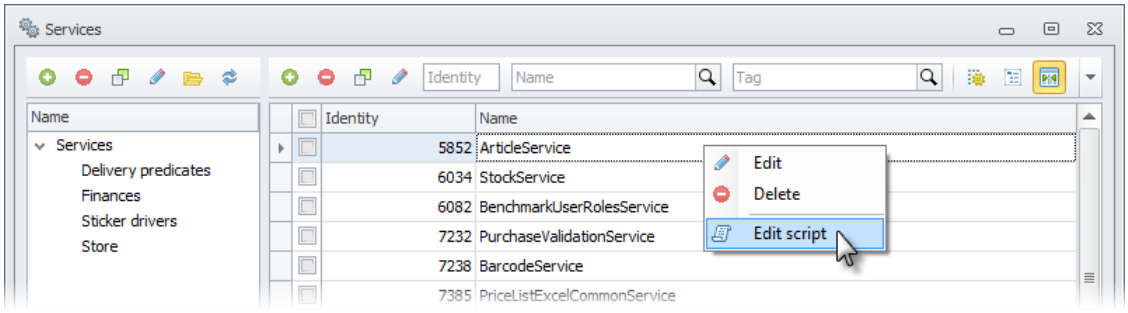
- *Full text search* – the script contains full compliance to the searched text;
- *All of the words* – the script contains all the searched words;
- *At least one word* – the script contains at least one of the searched words;
- *Does not contain words* – the script does not contain any of searched words;
- *Tag* – the script is marked with the tag.
- *In scripts text* – search in the text of scripts is carried out with the set flag;
- *In descriptions* – search in the descriptions of scripts is carried out with the set flag;
- *In resources* – search in the resources of scripts is carried out with the set flag.

The filter is applied by clicking the key button "Apply filter". Lines with the found text (only for *Full text search* option) are highlighted in the form of a script preview:



Besides the dictionary of scripts the script can be opened:

- from the form of editing of its main part;
- from a list form of its main part, selected an item *Edit script* in a context menu, available by the right-click on the chosen record of the list form:



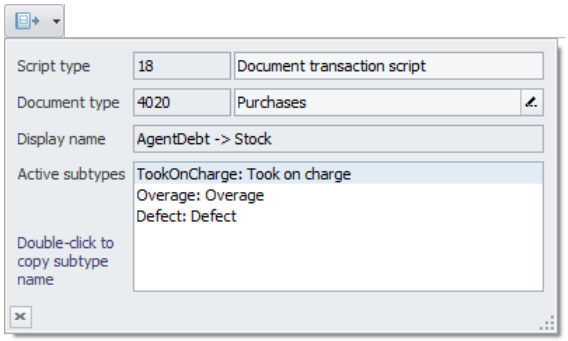
The name of its class and the identifier is displayed in the form heading of the script. All properties of the script are grouped in the form of editing in tabs:

Besides the identifier of an open script, the elements of management are located in the heading of script editing form, oriented to work with these tabs "Script text":

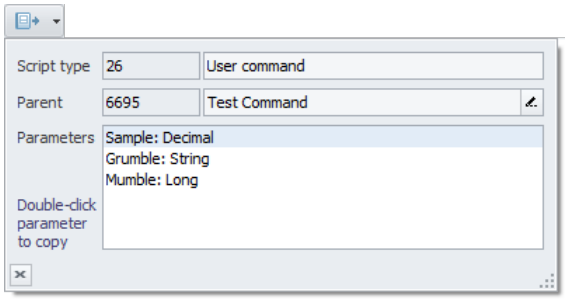
- key button – opens the form of editing of the main part of the script, corresponding to the type (*Type*) of the script:
  - Custom report – user report on a total;
  - Dictionary command – command over the dictionary record;
  - Dictionary event handler – form of dictionary editing;
  - Dictionary list command – command over the dictionary records;

- Document command – command over the document;
- Document event handler – editing form of the document type;
- Document list command – command over the documents;
- Interface – interface;
- Print form – print form;
- Report column provider – editing form of the dictionary;
- Service – service;
- Task – task;
- Total driver – total driver;
- Total event handler – form of total editing;
- Transactions – editing form of the document type;
- Transactions validators – form of total editing;
- User command – user command;
- Web service – web service.

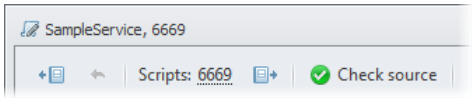
by clicking the arrow to the right of the key button  an information panel is opened in which additional data on the script and its main part are displayed. For example, for scripts of carrying there is a list of subtypes in which this script of carrying is included. By double left-click on the line of the list the name of the subtype is copied to the clipboard as *DocumentName.Subtypes.SubtypeName*.



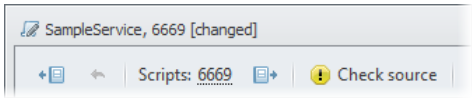
A list of script parameters is displayed for interactive commands and print forms. By double left-click on the line of the list the parameter name (in quotation marks) is copied to the clipboard;



- the "Check source" key button - check the script code for existence of the errors:
  - the icon will inform about the absence of errors found during the verification of the script code  ;

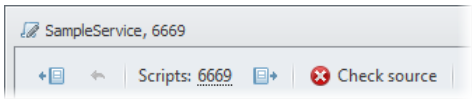


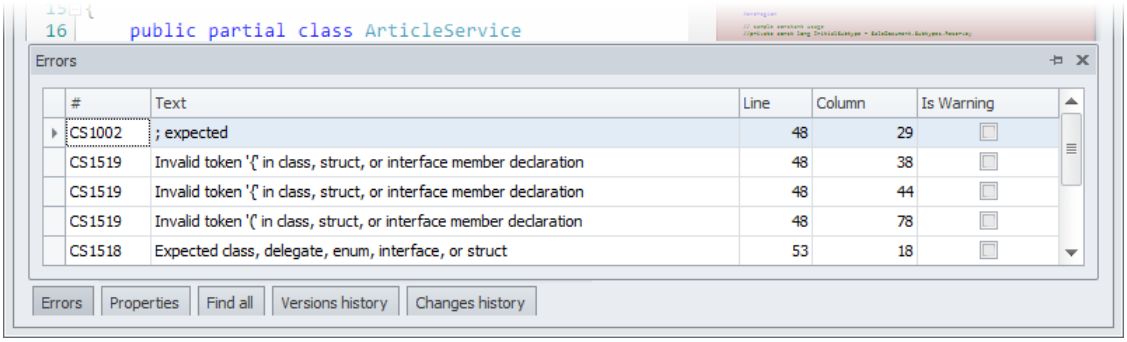
  - the following icon will inform about existence of the changes brought into the script code and untested changes  .



Click on the "Check source" key button with such icon generates and compiles the script. At the same time editing of the script text and its resources is not available;

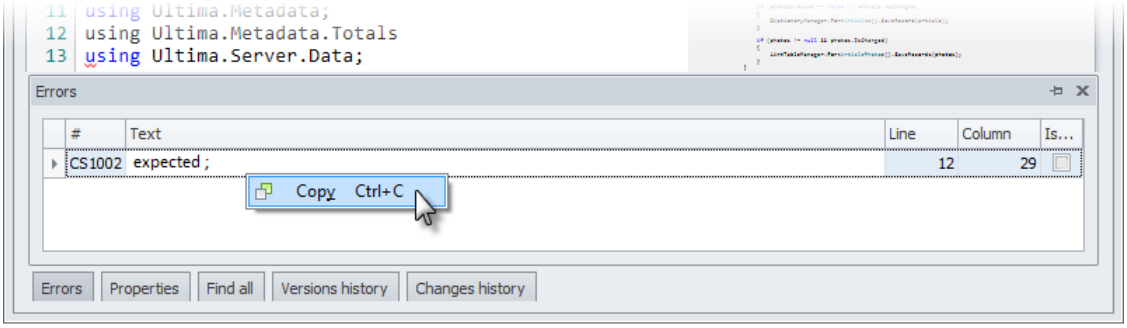
- finally, the icon will inform about the errors found during the verification of the script code . At the same time, the found errors will be displayed in the lower part of the form at the pop-up window:





Double left-click on the error leads to the positioning of the cursor on the line of the script to this error.

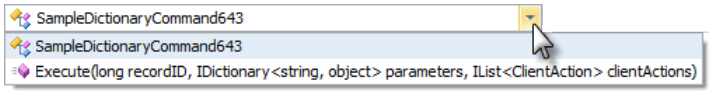
The text of the chosen error can be copied through the context menu, available by right-click or by pressing of a combination of keys Ctrl + C:



The error window can be opened at any time by left-click on the "Errors" key button in the lower left corner of the script editing form. Thus, the error window is automatically hidden, when you click the mouse in any area outside of it. In order to the error window will not be hidden, it should be fixed .

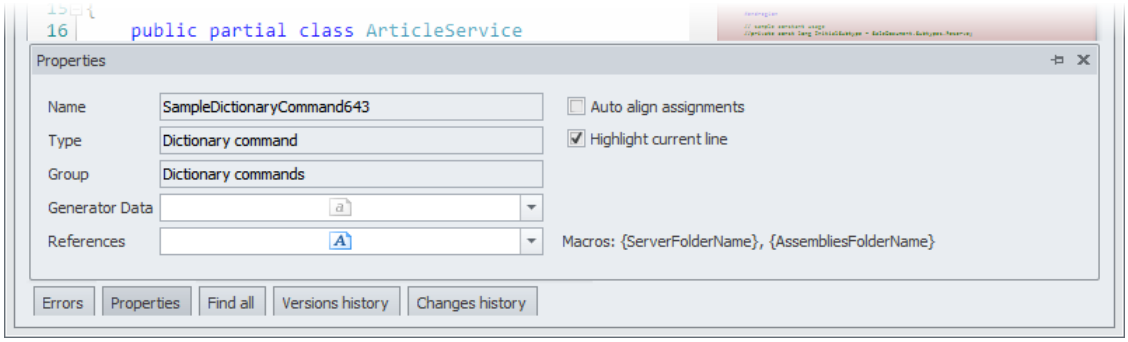
When mouseover at the "Errors" key button (without click) the error window will be opened, but will automatically disappear as soon as the mouse cursor leaves the "Errors" key button or an area of the error window;

- key buttons and – allow moving back and forth respectively in the script text on a history of cursor positions;
- a control element "Go to line" allows passing to the set line of the script code. For transition it is necessary to add a line number into a text field and to click the key Enter;
- a control element "Current element" performs two functions:
  - It displays on which element of the script code the cursor is set at the moment;
  - allows passing to the chosen code element:

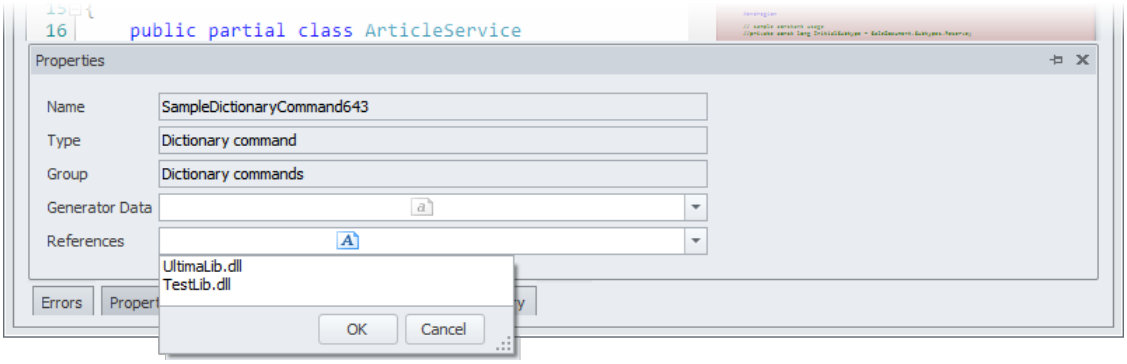


At the tab "Script text" there is an editor of the script code. In the right area of the editor the field of navigation it is placed, showing which part of the script is located on the screen:

- the "Properties" key button in the lower left corner of the edit form - opens a pop-up window below the script properties:



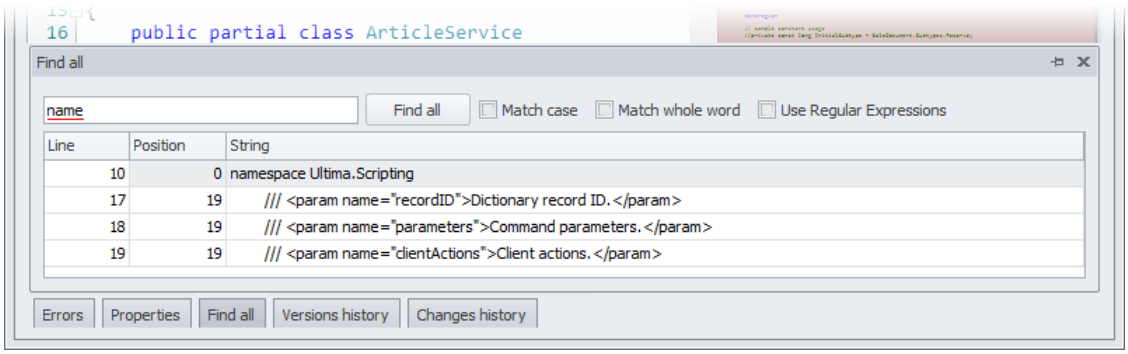
  - *Name* – script name is automatically assigned, and coincides with the class name of the script;
  - *Type* – script type is assigned automatically when creating a script;
  - *Group* – a group, which the script belongs to, is set in the main part of the script;
  - *Generator data* – an additional script parameter, used in its generated part;
  - *Reference* – additional assembly (external libraries), used by the script:



    The listed libraries have to be in GAC (information about Global Assembly Cache can be found on the website eng/rus), or in the specified directory *ServerFolderName* of the application server;
  - *Auto align assignments* – with the set flag there is an automatic alignment of the variables initialized in the script code;
  - *Highlight current line* – with the set flag the current line is highlighted in the editor;

  Thus, the property window is automatically hidden, when you click the mouse in any area outside of it. In order to the property window will not be hidden, it should be fixed .

  When mouseover at the “Properties” key button (without click) the error window will be opened, but will be automatically hidden as soon as the mouse cursor leaves the “Properties” key button or an area of the property window;
- the “Find all” key button in the lower left corner of the edit form - opens a pop-up window of search in the script text below:



  The search options, established by flags from the right of the "Find all" key button, allow to search:

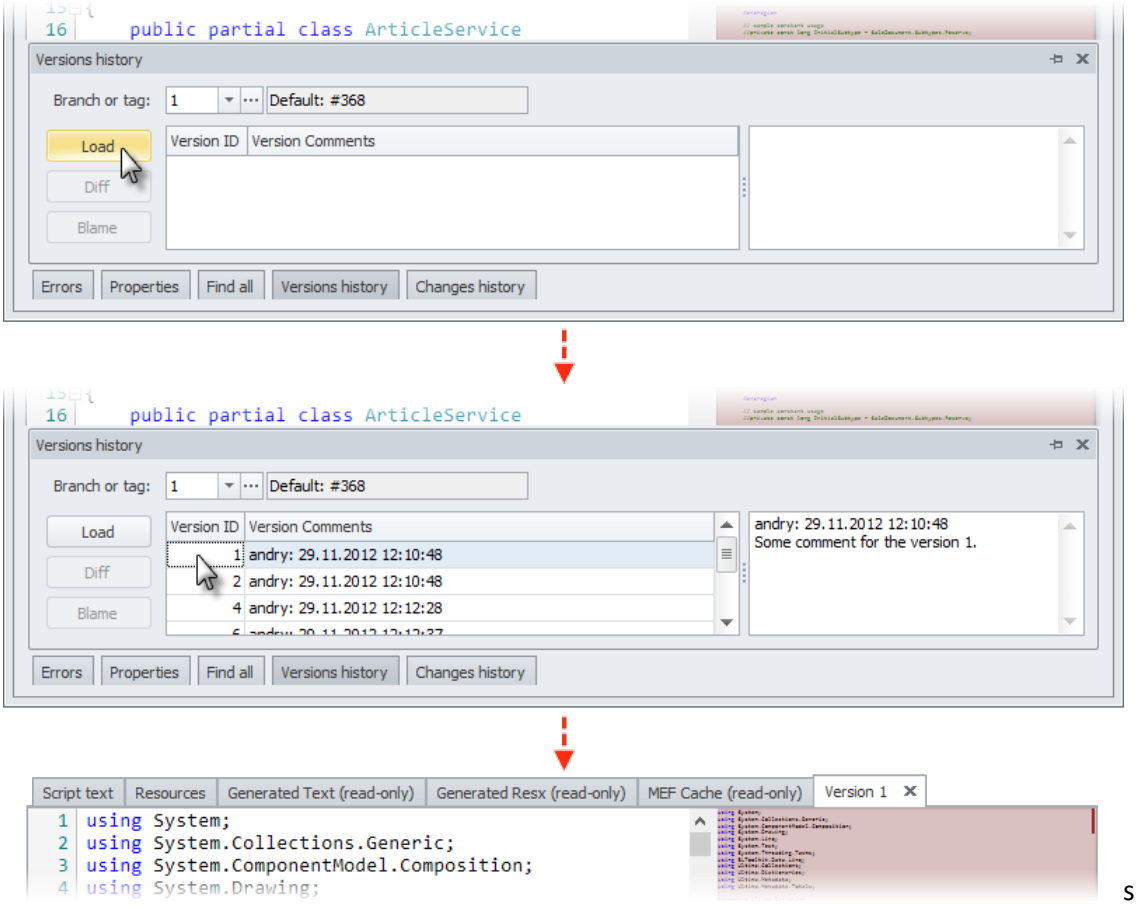
- *Match case* – match case;
- *Match whole word* – only the whole word;
- *Use Regular Expressions* – using regular expressions.

Double left-click on the search result leads to the positioning of the cursor on the line of the script with the found fragment.

Thus, the search window is automatically hidden, when you click the mouse in any area outside of it. In order to the search window will not be hidden, it should be fixed .

When mouseover at the “Find all” key button (without click) the search window will be opened, but will be automatically hidden as soon as the mouse cursor leaves the “Find all” key button or an area of the search window;

- the “Versions history” key button in the lower left corner of the edit form - opens a pop-up window with the script change history from version to version in the configuration branch, marked by the tag (branch or usual) chosen in the field *Branch or tag*:

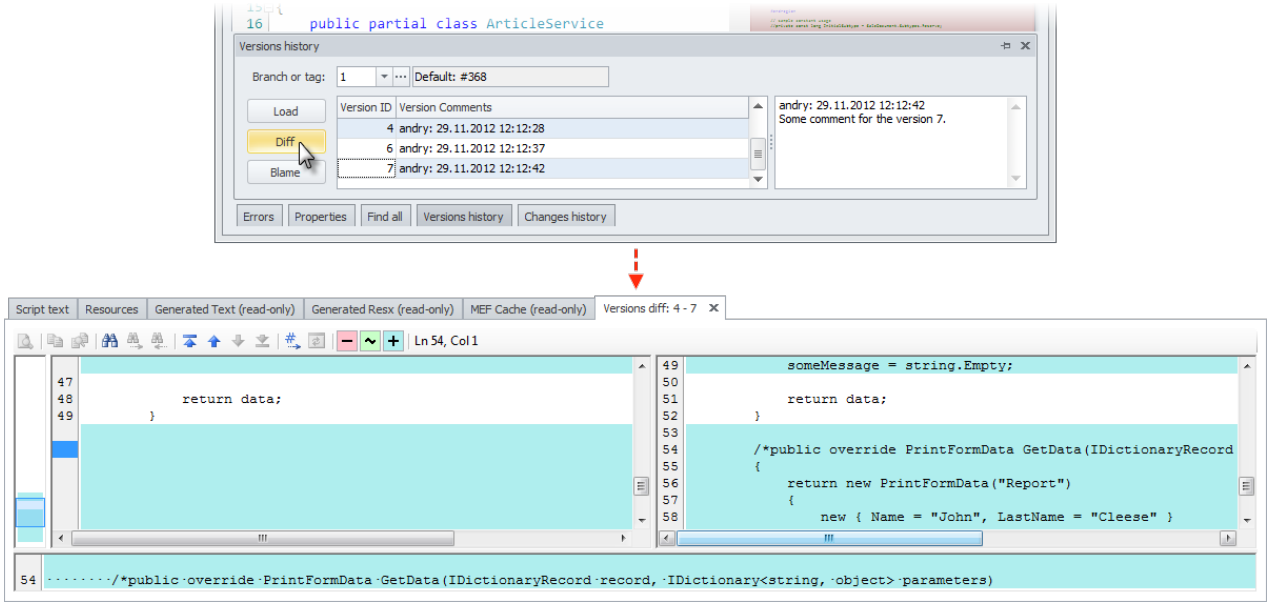


s

The history of changes according to versions is loaded by left-click on the "Load" key button. The latest version of the script in the current version of the configuration, opened in the "Script" tab, is also present in the history of changes.

The double left-click on the script leads to its opening in a new tab with the corresponding heading.

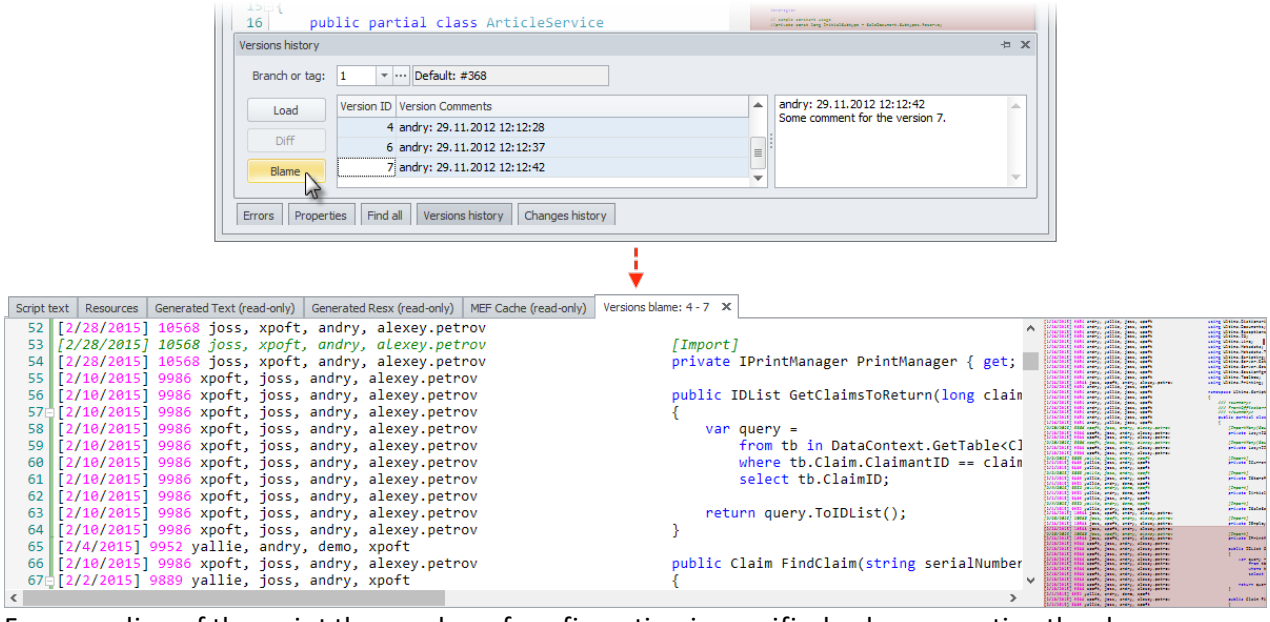
If to highlight, holding the Shift or Ctrl key pressed, two versions of the script and to press the "Diff" key button, it is possible to see the history of changes in details, which will be opened in a new tab with the corresponding heading:



In the lower part of the tab of the history of changes for the versions two versions of the line are shown for comparison, on which the cursor is set:

- light green highlights the changed lines;
- purple highlights a deleted text;
- blue highlights an added text.

If to highlight, holding the Shift or Ctrl key pressed, two and more versions of the script and to press the "Blame" key button, it is possible to see the changes that were made to the script in a new tab with the corresponding heading in what version of a configuration:

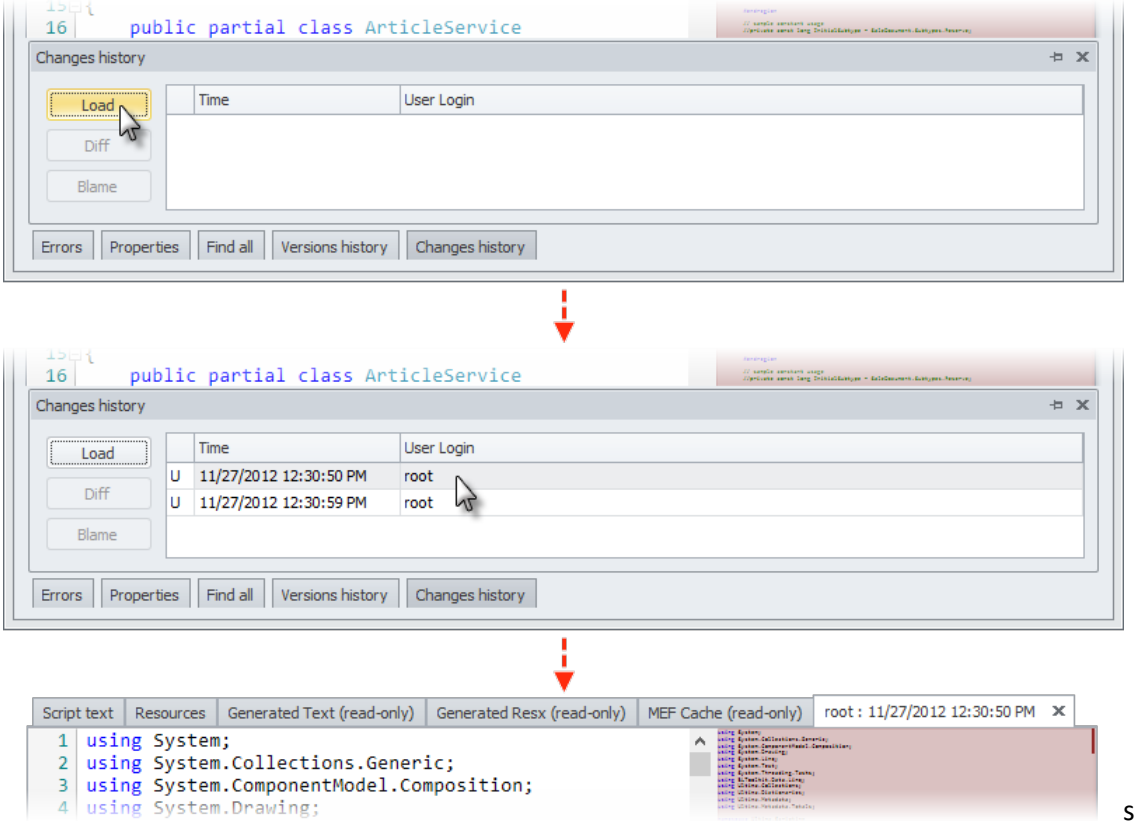


For every line of the script the number of configuration is specified, when operating the changes were made in it.

The pop-up window of the history of changes according to versions is automatically hidden, when you click the mouse in any area outside of it. In order to the change history window will not be hidden, it should be fixed .

When mouseover at the “Versions history” key button (without click) the window of change history according to versions will be opened, but will be automatically hidden as soon as the mouse cursor leaves the “Versions history” key button or an area of the window of change history according to versions;

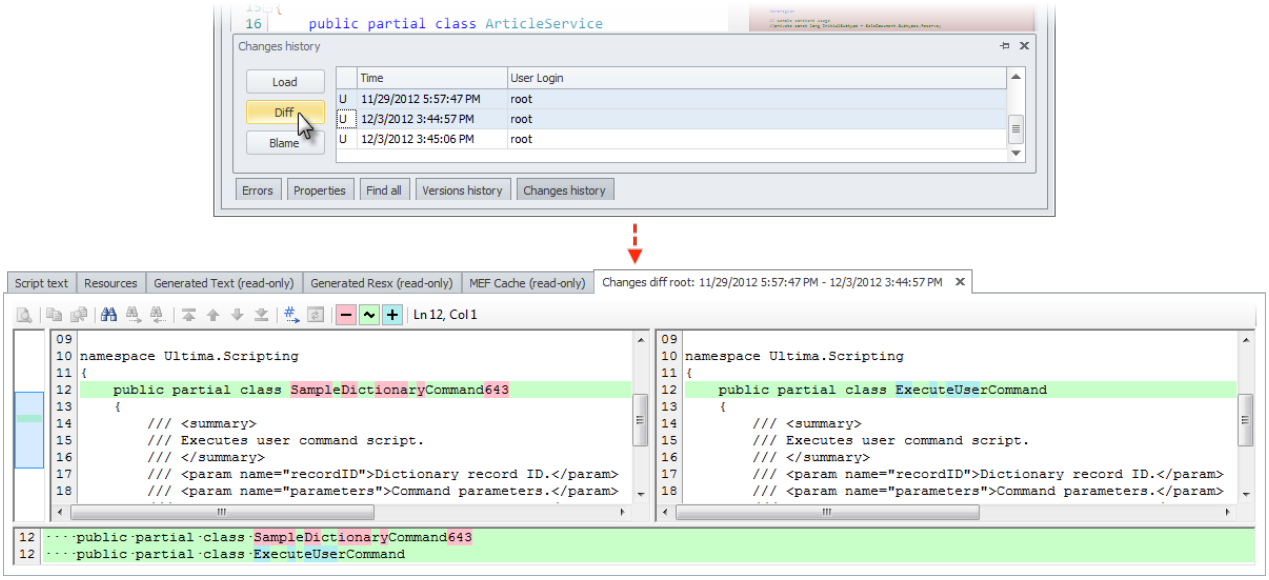
- the "Changes history" key button in the lower left corner of the edit form - opens a pop-up window with the of the script history changes at the current version of the configuration (with which the client application works):



s

The history of changes is loaded by left-click on the "Load" key button. The last saved version of the script, opened in the "Script" tab, is also present in the history of changes.
The double left-click on the script leads to its opening in a new tab with the corresponding heading.

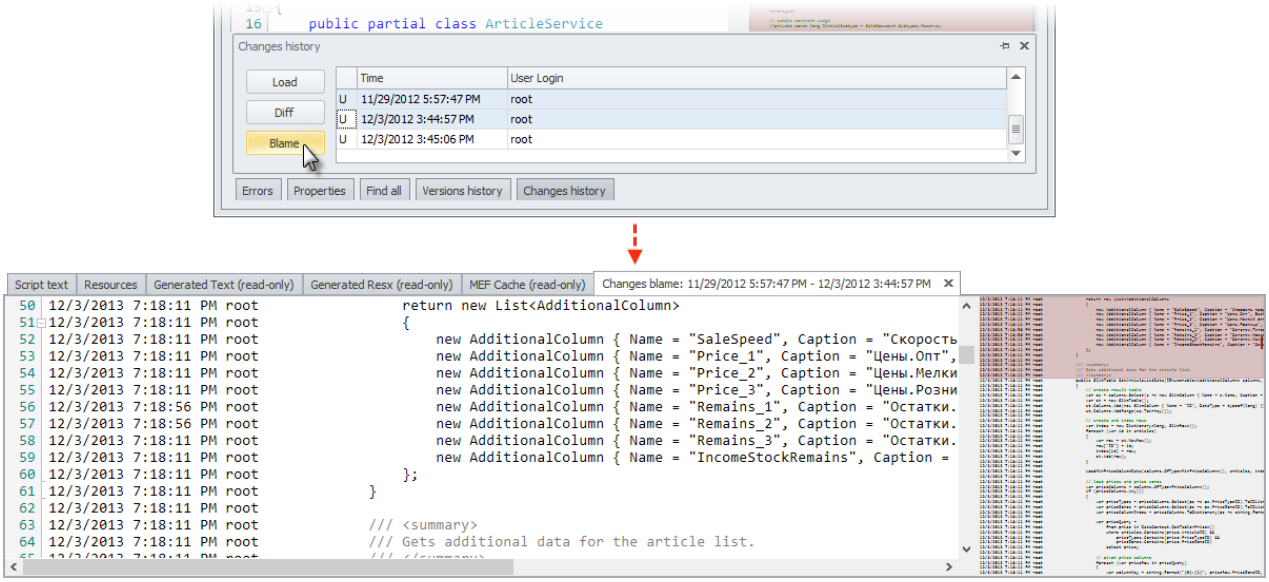
If to highlight, holding the Shift or Ctrl key pressed, two versions of the script and to press the "Diff" key button, it is possible to see the history of changes in details, which will be opened in a new tab with the corresponding heading:



In the lower part of the tab of the history of changes two versions of the line are shown for comparison, on which the cursor is set:

- light green highlights the changed lines;
- purple highlights a deleted text;
- blue highlights an added text.

If to highlight, holding the Shift or Ctrl key pressed, two and more versions of the script and to press the “Blame” key button, it is possible to see who and when made the changes in the script in a new tab with the corresponding heading:

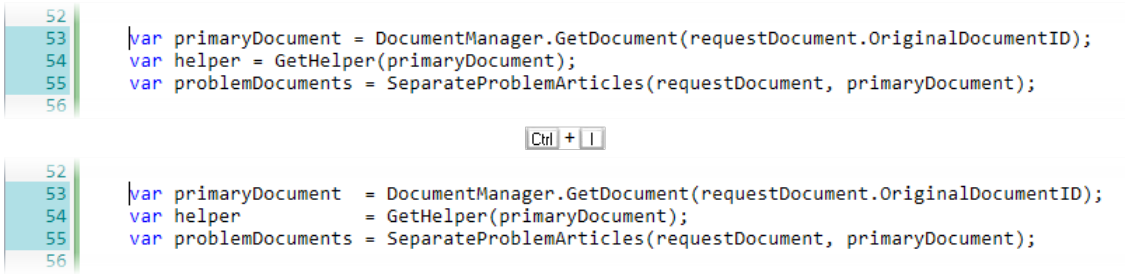


Each line of the script has the date of the introduction of changes and under what user they were made.

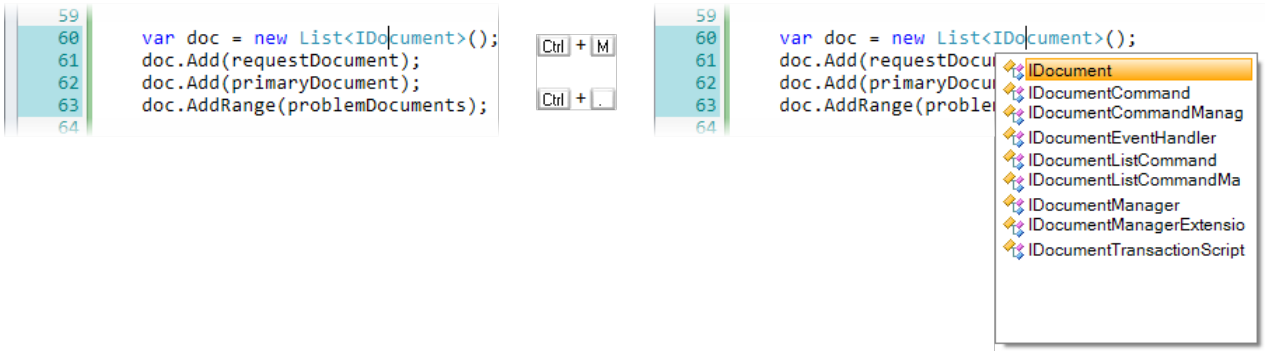
The pop-up window of the history of changes is automatically hidden, when you click the mouse in any area outside of it. In order to the change history window will not be hidden, it should be fixed . When mouseover at the “Changes history” key button (without click) the window of change history will be opened, but will be automatically hidden as soon as the mouse cursor leaves the “Changes history” key button or an area of the window of change history.

The editor of the script is realized in the following functionality and the following hot keys are supported:

- Ctrl + F – search dialog;
- F3 – to find the following occurrence of a required fragment (at the closed search dialog window);
- Ctrl + 0 – to return standard font size (font size is changed by the rotation of a mouse wheel when pressing the key button Ctrl);
- Ctrl + B – to set a breakpoint (breakpoint);
- Ctrl + N – to pass to the next line with the set breakpoint;
- Ctrl + Shift + N – to pass to the previous line with the set breakpoint;
- Ctrl + Shift + B – to remove the breakpoint;
- Ctrl + G – to pass to the specified line;
- Ctrl + H – to open the dialog of search and replacement;
- Ctrl + I – automatic space:

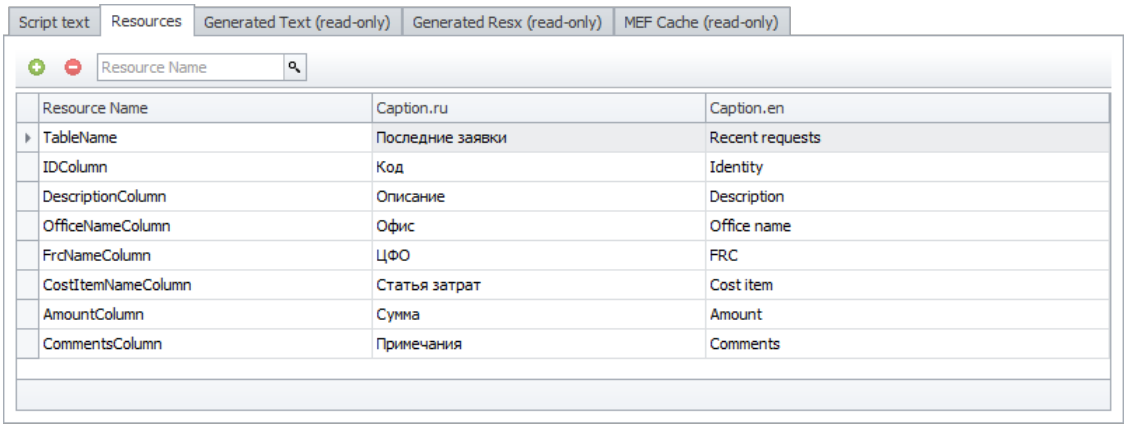


- Ctrl + M or Ctrl + . – use of technology *IntelliSense*:



- Ctrl + U – to transfer all symbols of the highlighted script fragment to the upper case;
- Ctrl + Shift + U – to transfer all symbols of the highlighted script fragment to the lower case;
- Ctrl + Shift + C – to comment the current line / the highlighted lines;
- Ctrl + M + O – collapse all code blocks.

In the tab "Resources" there are resources of the script and their value system localized for all languages:



Resource Name	Caption.ru	Caption.en
TableName	Последние заявки	Recent requests
IDColumn	Код	Identity
DescriptionColumn	Описание	Description
OfficeNameColumn	Офис	Office name
FrcNameColumn	ЦФО	FRC
CostItemNameColumn	Статья затрат	Cost item
AmountColumn	Сумма	Amount
CommentsColumn	Примечания	Comments

Resources are used if it is necessary to localize elements used in script. For example, if it is necessary to give a message to the user in his language.

For each of the resources a property will be generated (which can be found in the tab "Generated Text"), which value will be the text in the user's language.

Resources tab has its own toolbar, which in addition to the creation and deletion of resources allows filtering by *Resource Name* property. To filter the resources, supply the search string and click the search button. To reset the filter, clean up the search string and repeat the search.

Resources grid supports copying and pasting the resources via the standard Ctrl + C and Ctrl + V hotkeys.

There is an example of use of resources for localization of the form generated by the script with the table in which results of its performance are displayed:

```
var table = new SlimTable("RecentRequests") { query.ToArray() };
table.TableName = TableName;
table.Columns["ID"].Name = IDColumn;
table.Columns["Description"].Name = DescriptionColumn;
table.Columns["Comments"].Name = CommentsColumn;
table.Columns["Amount"].Name = AmountColumn;
table.Columns["OfficeName"].Name = OfficeNameColumn;
table.Columns["FrcName"].Name = FrcNameColumn;
table.Columns["CostItemName"].Name = CostItemNameColumn;
```

Script resources can be used both in the main class and in auxiliary one:

```
class MyUserCommand
{
	//the localized message from the resources (in the generated part of the script)
	internal static string Message {
		get { return ResourceManager.GetString("Message"); } }

	public void Execute()
	{
		//let's take the message resource from the current class
		throw new UltimaException(Message);
	}
}

class HelperClass
{
	void MyMethod()
	{
		//let's take the resource from the main class in which they are announced
		throw new UltimaException(MyUserCommand.Message);
	}
}
```

In the tab "Generated Text" there is a part of the script, automatically generated.

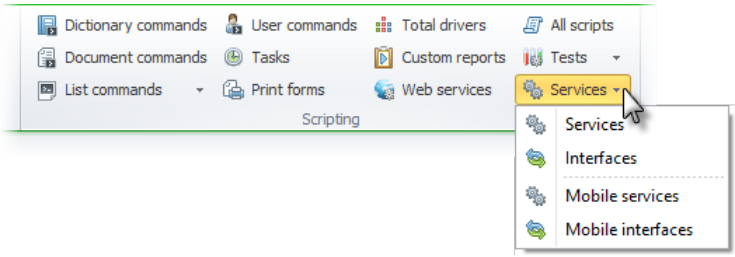
In the tab "Generated Resx" there is a generated code of resource file in the format resx - in the form in which the compiler will receive it.

In the tab “MEF cache” there is a cache of the library MEF (Managed Extensibility Framework), which detailed description of which can be found on the website MSDN eng/rus.

Services and interfaces

In the system Ultimate AEGIS® services are divided according to the destination into:

- *Services* – ordinary services;
- *Mobile services* – mobile services;
- *Web services* – web-services.



Services

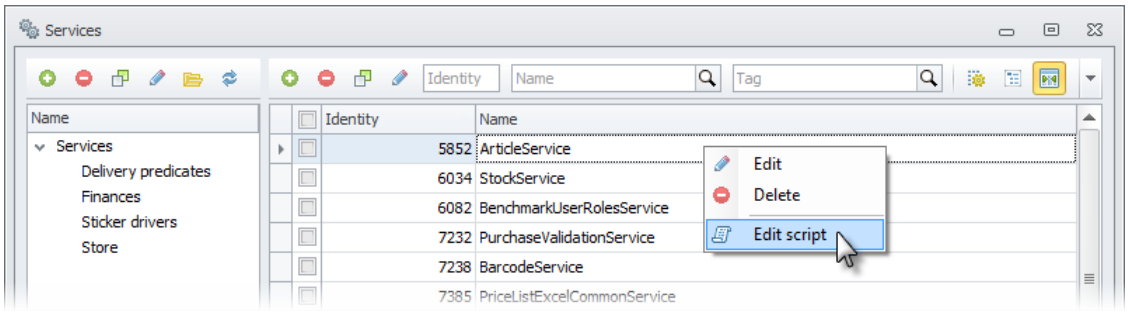
Services are the scripts realizing the specified interface. They are designed to store frequently used functionality.. When writing scripts there are often standard tasks which the application developer should solve time after time. Not to write a code of the same functionality again, developers often copy it from one script in another. In the situation when the functionality needs to be changed for any reason, the application developer have a task to find his code by all scripts where it is used, and everywhere to make the appropriate changes.

Services are also used to implement such standard functionality. Once having written the service for solving of typical task, you can resort to its functionality, while importing its implemented interface.

```
[Import]
private IServiceName ServiceName { get; set; }
```

Import is described in the section Use of Services in details.

The list of all interfaces can be found in the dictionary "Services":

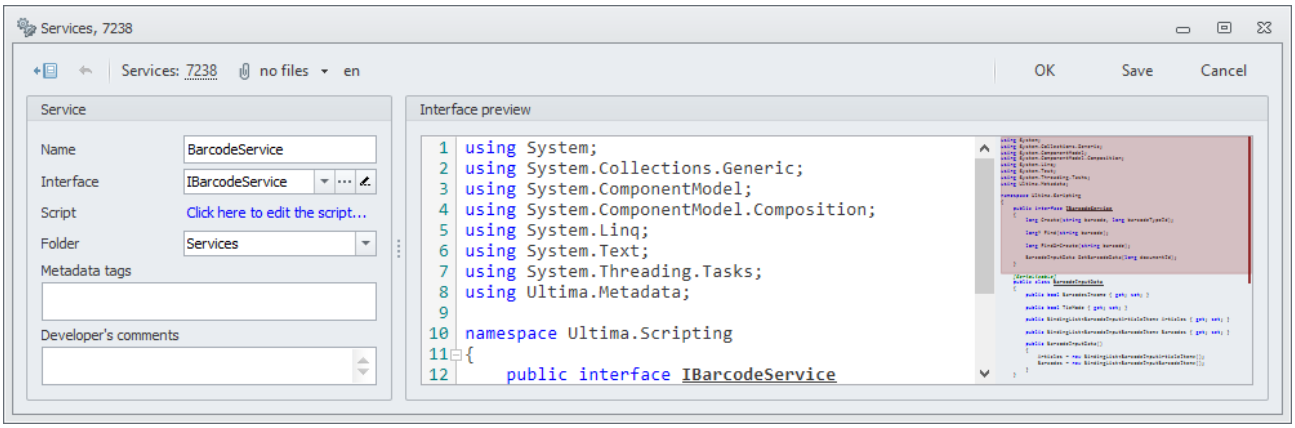


The dictionary window is divided into two parts: on the left, there is a tree of service groups, on the right – the list of service of the group selected on the left.

The dictionary records can be filtered by *Service name* (*Name*) and *Tags* (Tag).

The script of the service selected in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

The interface has the following properties:

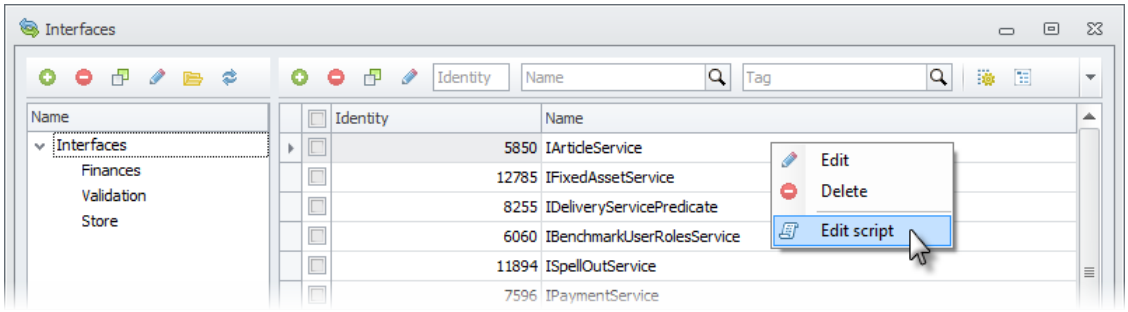


- *Name* – service class name. By default, the service name will be assigned with the interface name which it is created but without prefix “I”, and with number suffix (to guarantee uniqueness of names). For instance, for interface "IInterfaceName", a service will be created with the name "InterfaceName1233". The service name can be changed;
- *Interface* – an interface implemented with the service;
- *Script* – a link to service script. In case of creation of new service, the script is created automatically upon its saving. Click on the link *Click here to edit the script...* during creation of new mobile service will result saving of the service and its reloading, after that the script edit form will open;
- *Folder* – a group, which the service belongs to;
- *Metadata tags* – tags used for description of the service functionality;
- *Developer's comments* – comments of the application developer;
- *Interface preview* – a script of the interface implemented with the service. In the right area, the navigation area is located showing which part of the script is displayed on the screen.

Interfaces

 Interface – description of interactions of classes. Each service must implement some interface and, thus, the service becomes available for use. Moreover, one interface can be implemented with several services.

The list of all interfaces can be found in the dictionary "Interfaces":

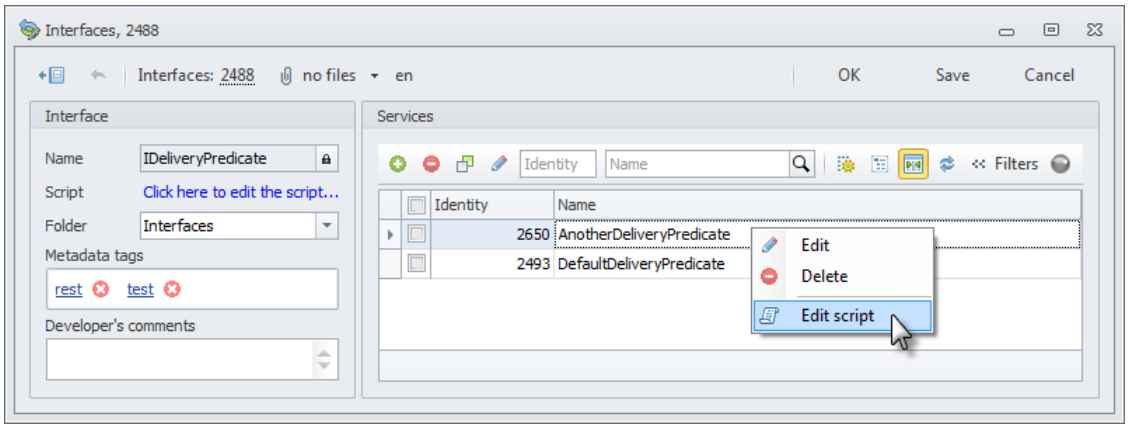


The dictionary window is divided into two parts: on the left, there is a tree of interface groups, on the right – the list of interface of the group selected on the left.

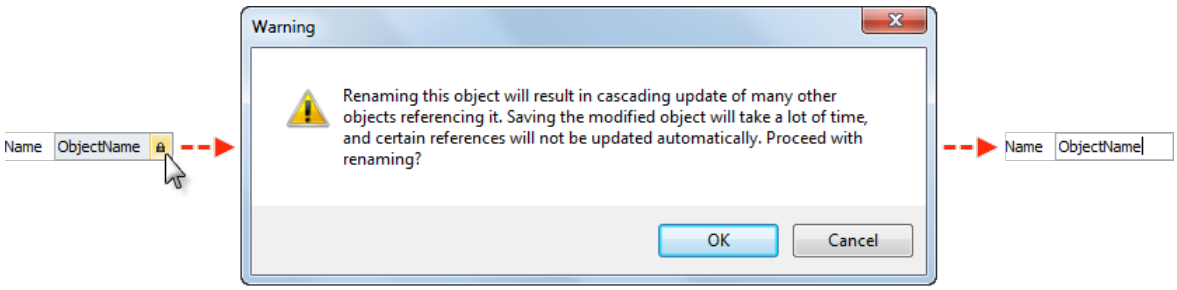
The dictionary records can be filtered by *Interface name* (*Name*) and *Tags* (*Tag*).

The script of the interface selected in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

The interface has the following properties:



- *Name* – interface class name. The interface name must have prefix "I", which is added automatically. Therefore, the entered interface name "NameOfInterface" will be automatically converted into "INameOfInterface". If necessary, the name can be changed:



- *Script* – a link to interface script. In case of creation of new interface, the script is created automatically upon its saving. Click on the link *Click here to edit the script...* during creation of new interface will result into saving of the interface and its reloading, after that the script edit form will open;
- *Folder* – a group the interface belongs to;
- *Metadata tags* – tags used for description of the interface functionality;
- *Developer's comments* – comments of the application developer;
- *Services* – a list of services implementing particular interface.
  The list of services can be filtered by the Service *name(Name*).
  The services can be created or removed using corresponding buttons in the toolbar:
  - in case of service creation, its edit form will be opened. By default, the service will be assigned with the interface name without prefix "I", but with number suffix (to guarantee uniqueness of names). For instance, for interface "INameOfInterface", a service will be created with the name "NameOfInterface1234". The service name can be changed;
  - in case of service removal, it will be removed not only from the list of service implementing particular interface but from dictionary of services.

  The service can be opened in the edit form by double-click of the left mouse button on it in the list.
  The script of the service selected in the edit form can be opened directly from the list of services, having selected item *Edit script* in the context menu.

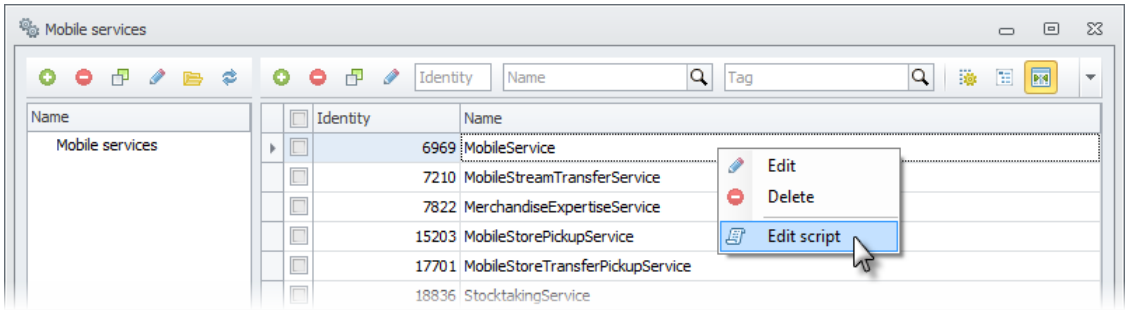
Mobile services

For integration with mobile applications, Ultimate AEGIS® system offers the use of mobile services.

Mobile service implements the specified mobile interface, which describes in turn the classes applied for interaction of mobile client applications. Moreover, any primitive types of data can be used, which are available both on mobile application and on application server..

The client mobile application is developed in C# using Xamarin http://xamarin.com/ with employment of mobile interfaces, which are compiled into a separate library mobilemetadata.dll (ultimalib.dll and mobileinterfaces.dll libraries are required too).

The list of all mobile services can be found in the dictionary "Mobile services":

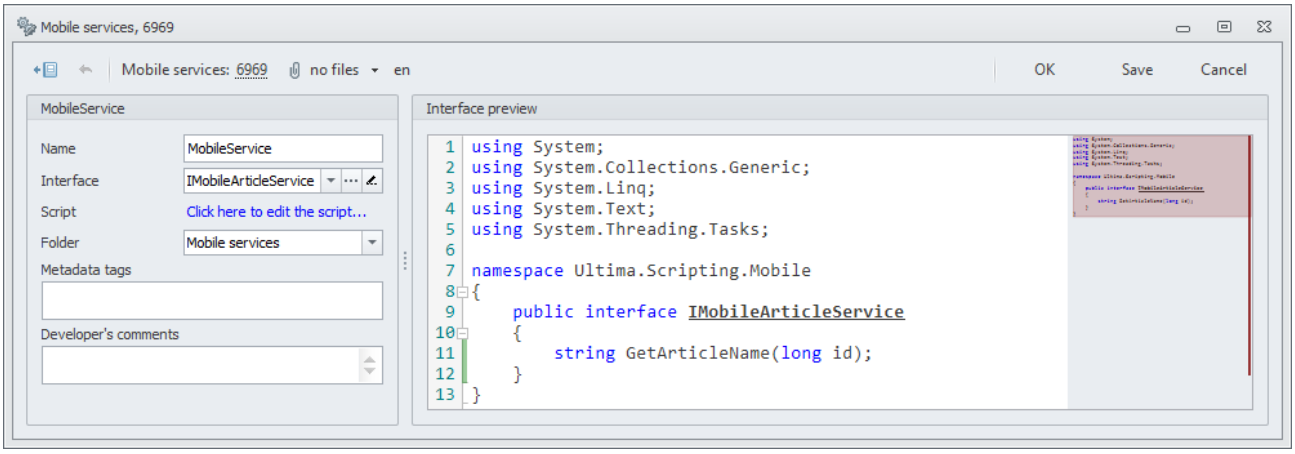


Dictionary window is divided into two parts: on the left, there is a tree of mobile services groups, on the right – the list of services of the group selected on the left.

The dictionary records can be filtered by *Mobile service class name* (*Name*) and *Tags* (Tag).

The script of the service selected in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

The mobile service has the following properties:



- *Name* – mobile service class name;
- *Interface* – an interface implemented with the service;
- *Script* – a link to service script. In case of creation of new service, the script is created automatically upon its saving. Click on the link *Click here to edit the script...* during creation of new mobile service will result saving of the service and its reloading, after that the script edit form will open;
- *Folder* – a group, which the service belongs to;
- *Metadata tags* – tags used for description of the service functionality;
- *Developer's comments* – comments of the application developer;
- *Interface preview* – a script of the interface implemented with the service. In the right area, the navigation area is located showing which part of the script is displayed on the screen.

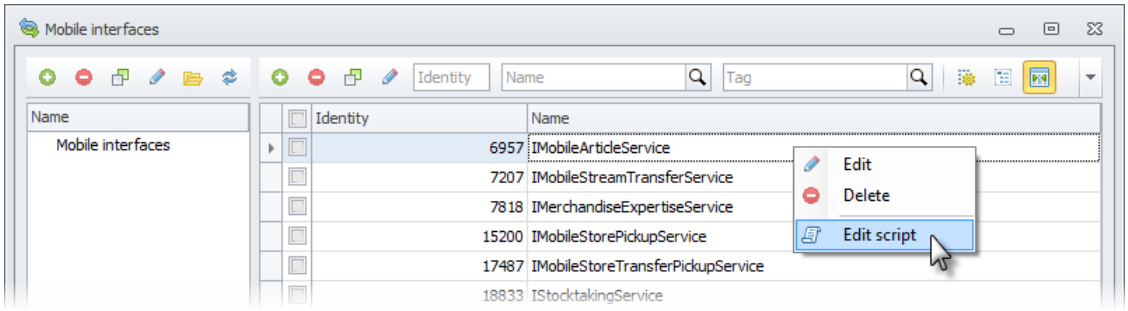
The mobile services are executed in the *protected* mode on behalf of the user that logged in the system

Mobile interfaces

Each mobile service must implement some mobile interface and, thus, the service becomes available for use.

Any primitive types of data, which are available both on mobile application and application server, can be used in mobile interfaces. Moreover, the metadata classes cannot be referred to.

The list of all mobile interfaces can be found in the dictionary "Mobile interfaces":

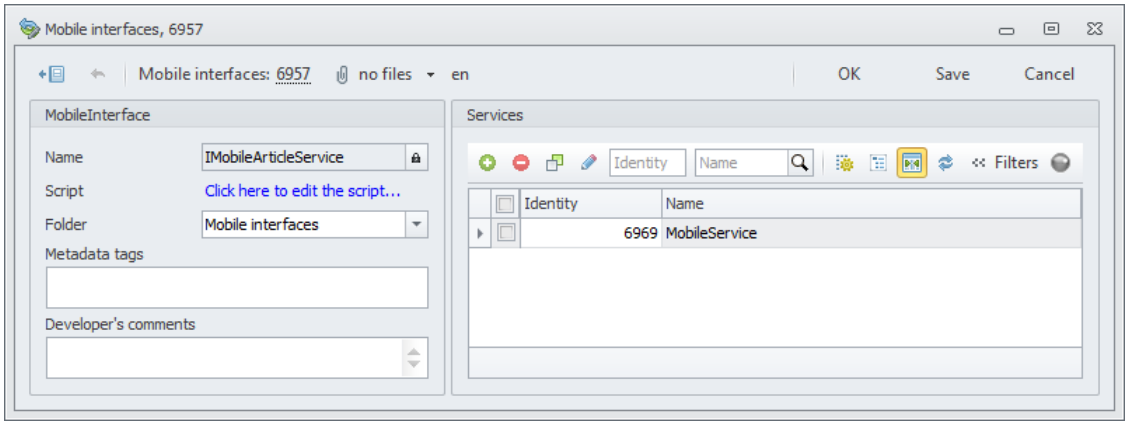


Dictionary window is divided into two parts: on the left, there is a tree of interface groups, on the right – the list of interface of the group selected on the left.

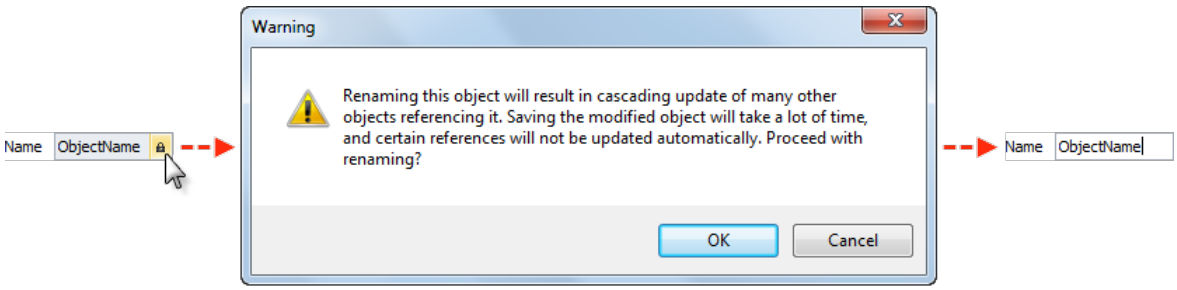
The dictionary records can be filtered by *Mobile interface name* (*Name*) and *Tags* (Tag).

The script of the mobile interface selected in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

The mobile interface has the following properties:



- *Name* – mobile interface name. If necessary, it can be changed:



- *Script* – a link to interface script. In case of creation of new interface, the script is created automatically upon its saving. Click on the link *Click here to edit the script...* during creation of new mobile interface will result saving of the interface and its reloading, after that the script edit form will open;
- *Folder* – a group the interface belongs to;
- *Metadata tags* – tags used for description of the interface functionality;

- *Developer's comments* – comments of the application developer;
- *Services* – a list of mobile services implementing particular interface.
  The list of services can be filtered by the *Name* of service (*Name*).
  The services can be added or removed using corresponding buttons in the toolbar: In case of service removal, it will be removed not only from the list of service implementing particular interface but from corresponding dictionary of mobile services.
  The service can be opened in the edit form by double-click of the left mouse button on it in the list.

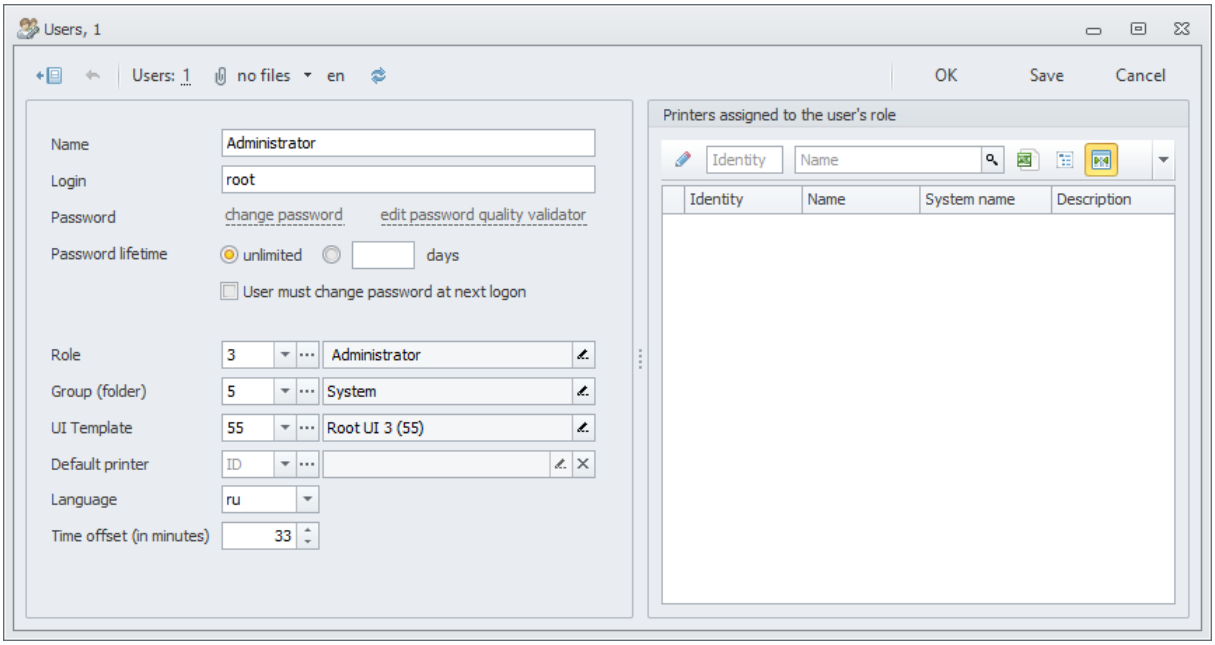
Kernel services

Kernel services are special scripts which are integrated with a kernel and provide additional functionality for the operations which are built in the system. Kernel services affect on the work not only applied, but also kernel operations. If mistreated they may disrupt the entire software package, therefore to work with them is recommended to entrust to the most skilled programmers. These scripts are:

- Common event handlers of all dictionaries and documents.
- Password quality checking service.

Password quality checking service

By default, the system makes no demands to the quality of the password (except that the password must be non-empty). To check the quality of the password it is possible to realize a kernel service, which will measure the quality of the password before its change. To access the service checks the password quality in the user dictionary there is a special link (edit password quality validator), located next to the link to change the password:



The interface of the password quality control service *IPasswordQualityValidator* includes the following methods:

- *GetPasswordQuality (string password)* – returns the quality of the transmitted password string on the following scale:
  - *None* – a blank password (the system does not allow blank passwords);
  - *Weak* – weak password;
  - *Average* – average password;
  - *Strong* – strong password;

- *Insane* – a password, exceeding requirements to the quality of the password;
- *CheckPasswordValidity (string password)* – checks whether it is possible to use the specified password in the system. The object *PasswordValidationResult* has to be result of the method, having the following properties:
  - *IsValid* – whether the password meets the system requirements;
  - *ErrorMessage* – a message explaining why the password does not meet the requirements (if it is necessary).

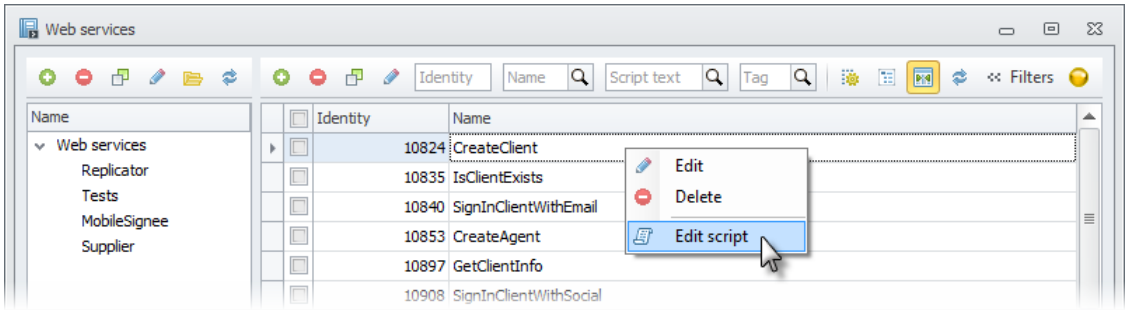
Web services

For integration with external applications, Ultimate AEGIS® system offers using web services according to SOAP protocols or with employment of REST API. The application is designed to process high load and therefore the development tools are specially optimized to create web services with message-based design. For convenience of external developers, serialization is supported in XML and JSON. The tools for creation of web services allow describing DTO (Data Transfer Object) in graphic interface for request and response value. Thus, each web service consists of:

- query class;
- response class;
- it is the script that receives the query and generates response.

The system hides the routing, serialization and de-serialization tasks from the application developer.

The list of all web services can be found in the dictionary "Web services":



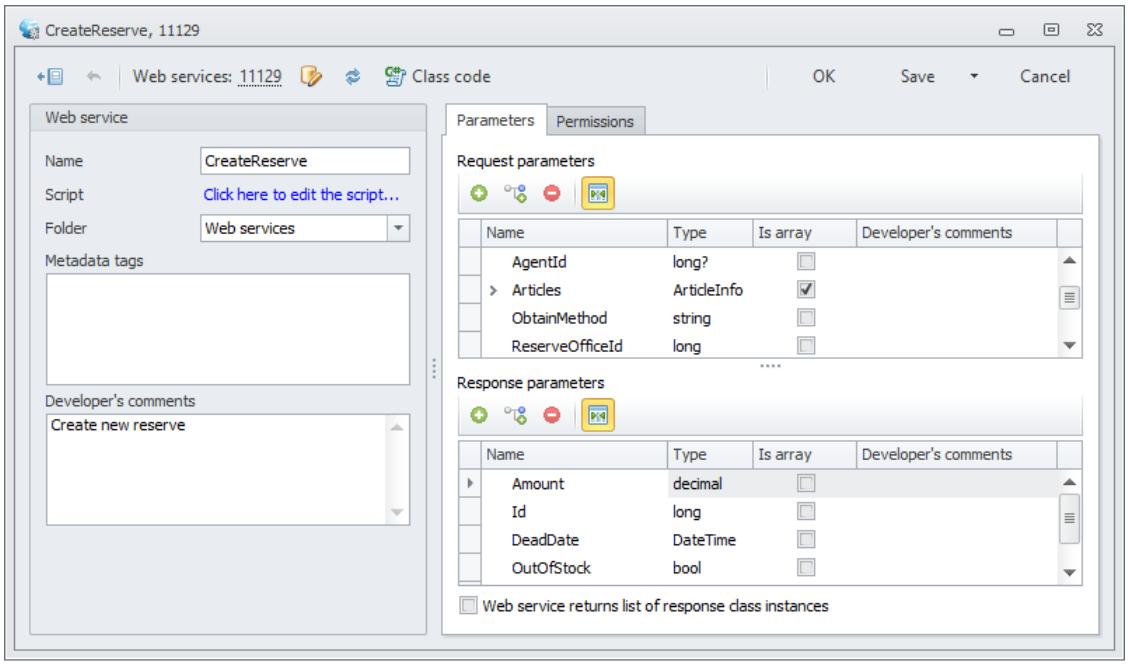
The dictionary window consists of two parts: a tree of the group of web services is displayed to the left, a list of services selected on the left of the groups is displayed to the right.

The dictionary records can be filtered by Web service class name (*Name*), its text (*Script text*) and *Tags* (*Tag*).

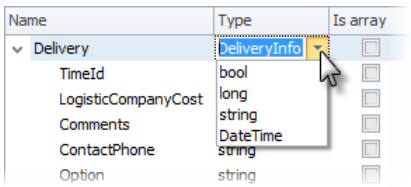
Cloning of metadata objects such as dictionaries, link tables, and web services is detailed in the Metadata cloning section.

The script of the service selected in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

The web service has the following properties:



- *Name* – web service class name;
- *Script* – a link to web service script. In case of creation of new service, the script is created automatically upon its saving;
- *Folder* – a group, which the web service belongs to. Click on the link *Click here to edit the script...* during creation of web service will result saving of the service and its reloading, after that the script edit form will open;
- *Metadata tags* – tags used for description of the web service functionality;
- *Developer's comments* – comments of the application developer;
- *Request parameters* – parameters of (DTO) *request*:
  - *Name* – parameter name;
  - *Type* – parameter type, POCO types (Plain Old CLR Object) can be used as the value – primitive CLR types, dates, terms. The type can be selected from existing ones or entered manually;
  - *Is array* – the parameter can be an array of specified type, if this flag is set;



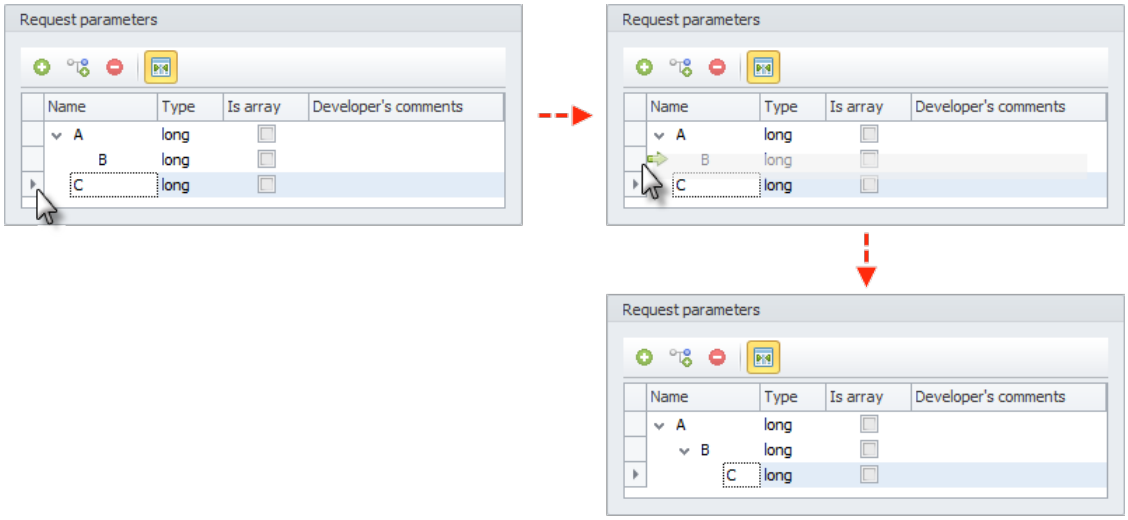
  - *Developer's comments* – comments of the application developer;
- *Response parameters* – parameters of (DTO) response *response*. Their features are similar to query parameters.

  Parameters of query and response can have complex nested structure. If parameter has child parameters, separate nested class will be generate. To create children parameter you need select a parent parameter and click in the toolbar of parameters.

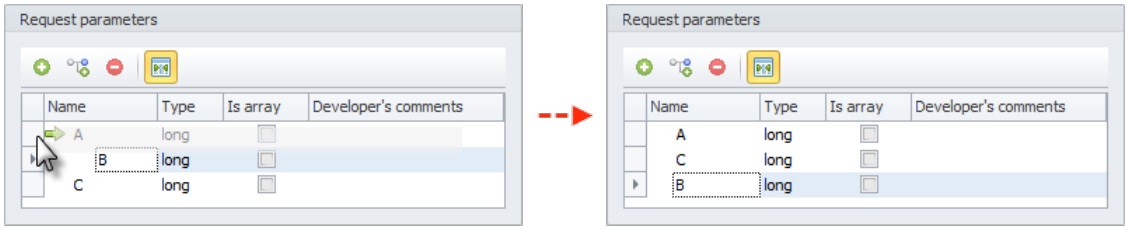
  In case of removing parent parameter, all child parameters will be removed.

  Parameters can be copied and pasted using the familiar Ctrl + C and Ctrl + V hotkeys.

Parameters in lists can be moved out of title bar by holding left mouse button. Thus, selected parameter may be child parameter for any other. Arrow indicate which parameter will be selected as parent:

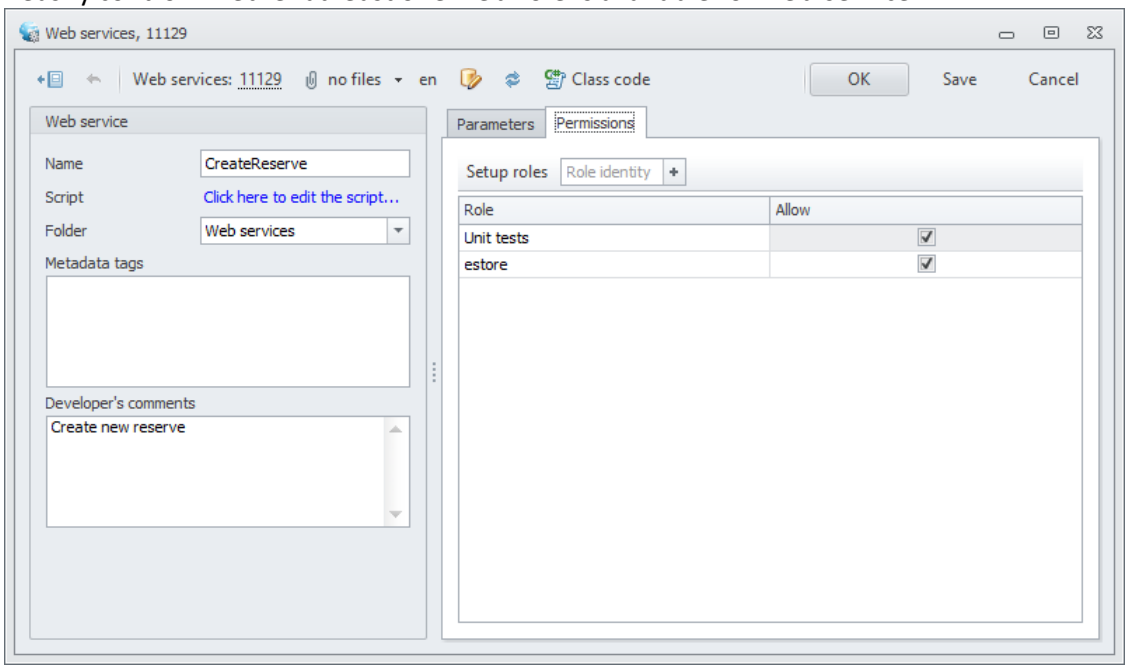


In case second-level parameter moves to his parent first-level parameter, it also moves to the first level:

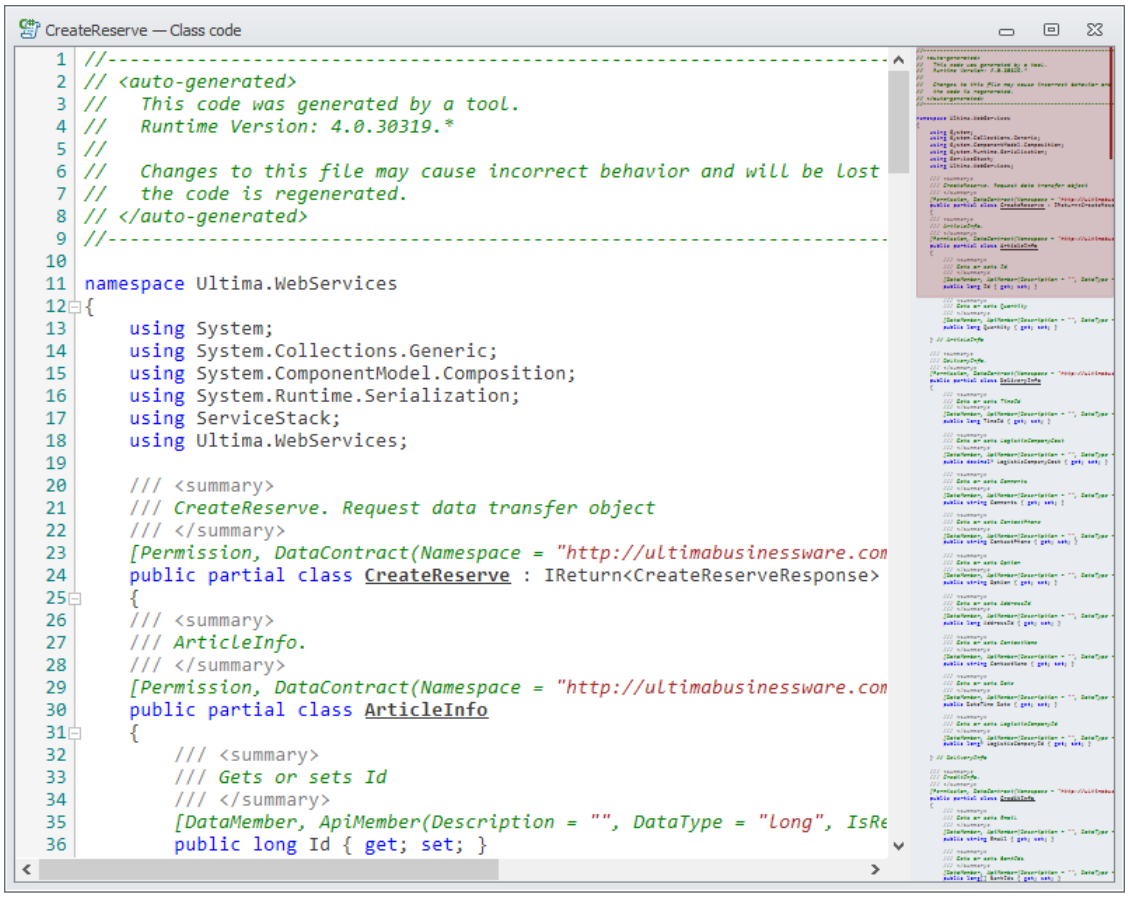


- *Web service returns list of response class instances* - flag which shows that web-service return value list.

The tab *Permissions* let quickly set up the web-roles list, which have access to this web-service. Roles list allow easily control whether at least one web-role is available for web-service:



By button tap "Class code" generated class, which describe web-service, is opened in toolbar of editing form:



Example of web-service MyOrders which return documents list in specified interval of dates:

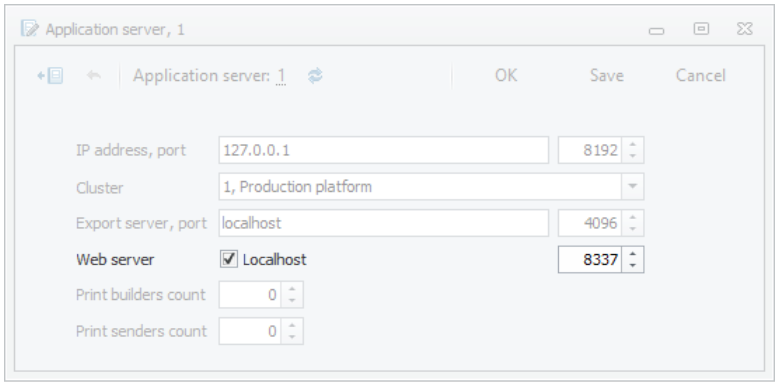
```
class MyOrders : IReturn<List<MyOrdersResponse>>
{
	DateTime DateFrom
	DateTime DateTo
}

class MyOrdersResponse
{
	int ID
	string Comments
	DateTime Date
	Article[] Articles

	class Article
	{
		int ID
		string Name
		decimal Price
	}
}
```

IWebService interface implements the following methods:

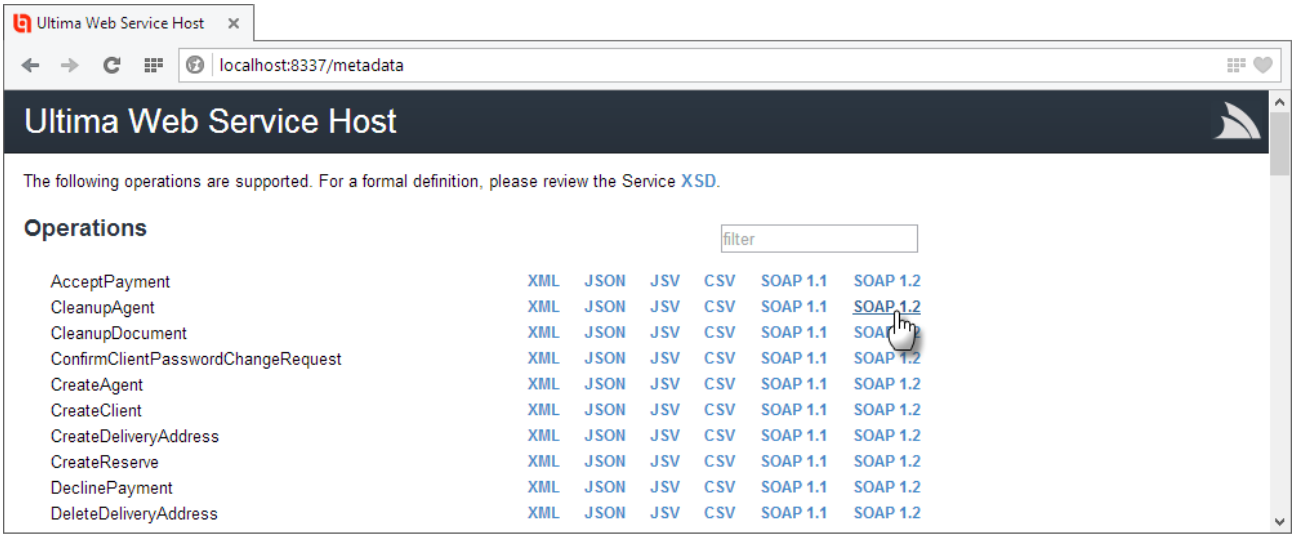
- *Get(TRequest request)* – executes specified GET-request;
- *Put(TRequest request)* – executes specified PUT-*request*;
- *Post(TRequest request)* – executes specified POST-*request*;
- *Delete(TRequest request)* – executes specified DELETE-*request*;

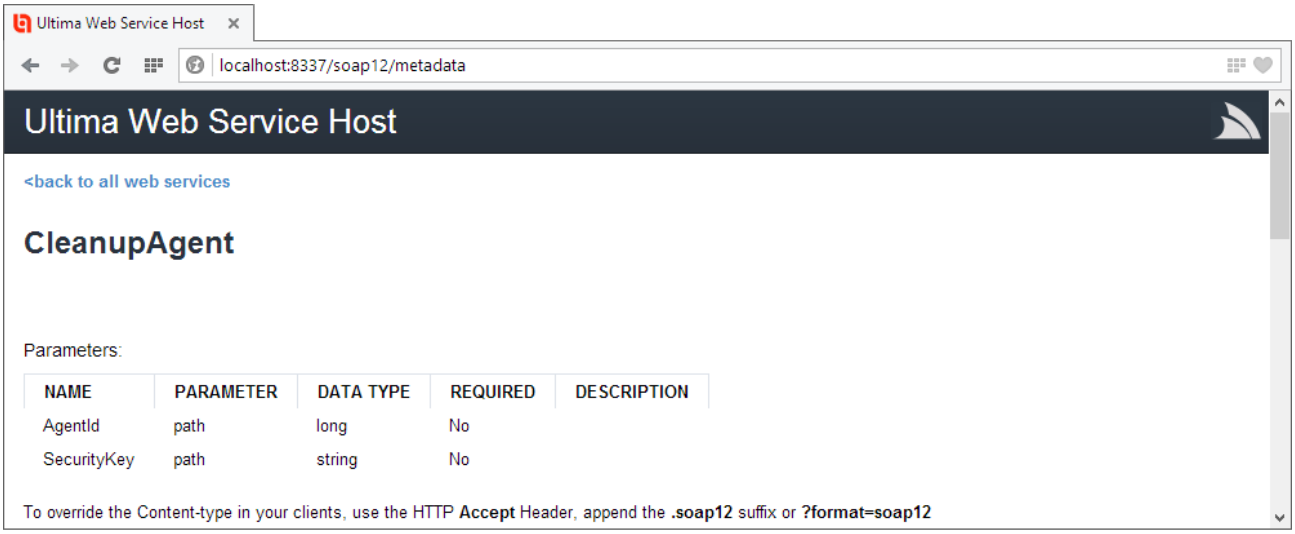
- *Options(TRequest request)* – executes specified OPTIONS-*request*;
- *Any(TRequest request)* – used for execution of the *request*, which method differs from specified above.

In addition to universally accessible web service, deployed by the administrator at particular address, it can be started locally. To do that, indicate *Localhost* in the configuration of application server as its address:

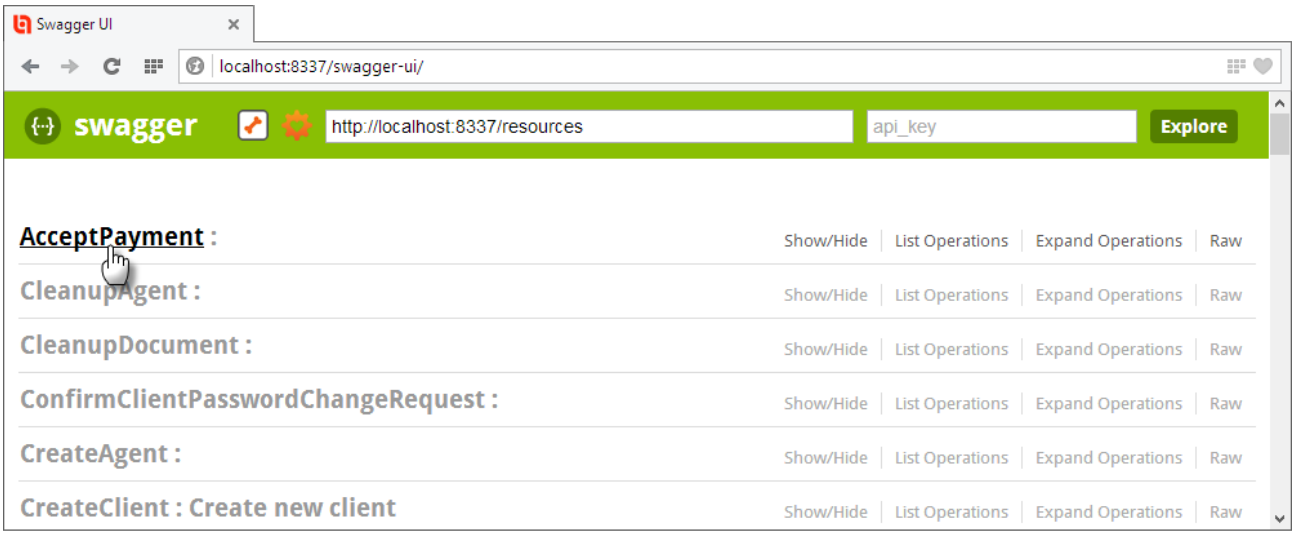


The web developer can receive a list of available web services at their address:

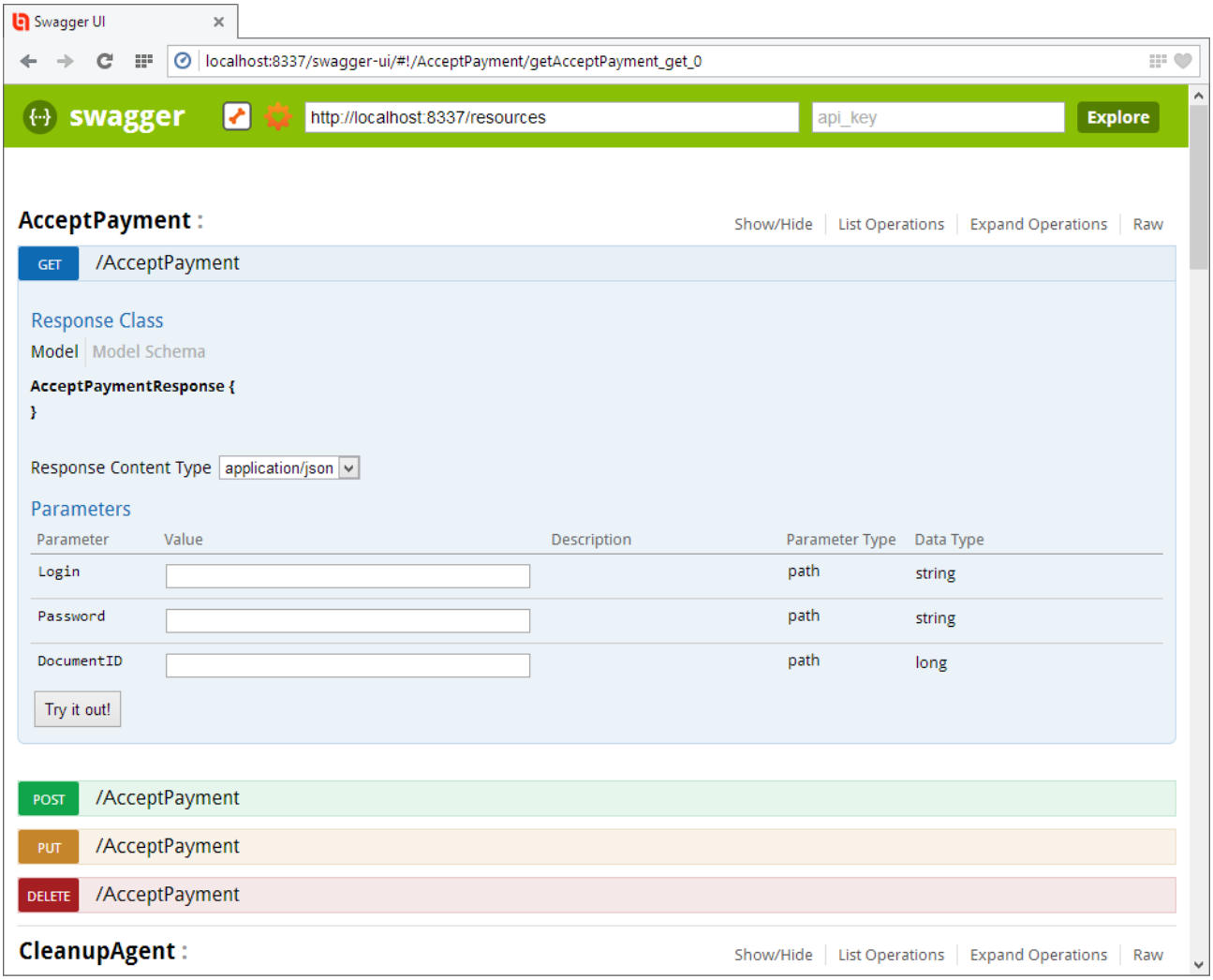




Swagger (specification, description and documentation of REST service) is supported too:

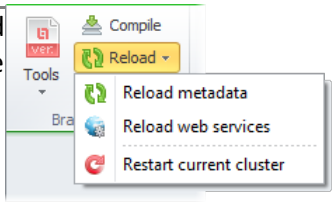


By selecting the service and operation type Get/Post (and, if necessary, having set the values of parameters), you can check the service functioning by clicking "Try it out!":



The requests to the web service are executed always on behalf of WebService user with their permissions in the *protected* mode.

After any changes made to web services, they should be reloaded with the command *Reload web services*, for these changes to take effect.



Access to web-services is limited and provided only to authorized users.

Such restriction is realized for security purposes that different applications could have access only to their web services through the corresponding users. In this case the compromise of one application will not affect on the other applications, which also use the web services.

Authorization for all web methods of the service is mandatory. For authorization HTTP Basic authentication is used.

In case of unsuccessful or incorrect authentication of the user of web service the HTTP code ***#401*** (Unauthorized).

In case when the user has no rights on this method HTTP code comes back ***#403*** (Forbidden).

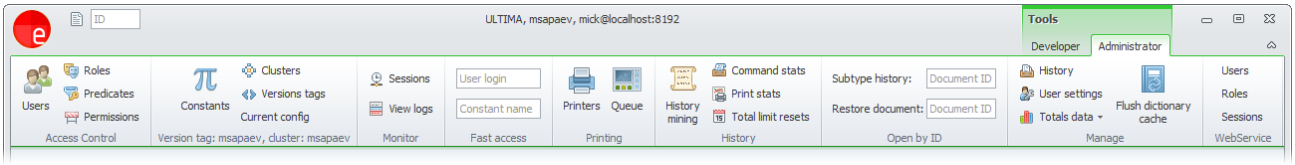
In order not to pass the authentication procedure permanently, it is necessary to save the cookie "ss-id", transmitted by web service, and pass it on requests.

In order to use authentication in SoapUI (http://www.soapui.org/), it is necessary to set *Authorization: Basic*, add *Username* and *Password* and choose *Authenticate pre-emptively*.

However, the authentication does not work with Soap 1.1 / Soap 1.2 / Xml clients of ServiceStack.

Granting of permissions for execution of web services is implemented by roles similarly to ordinary users of the system.

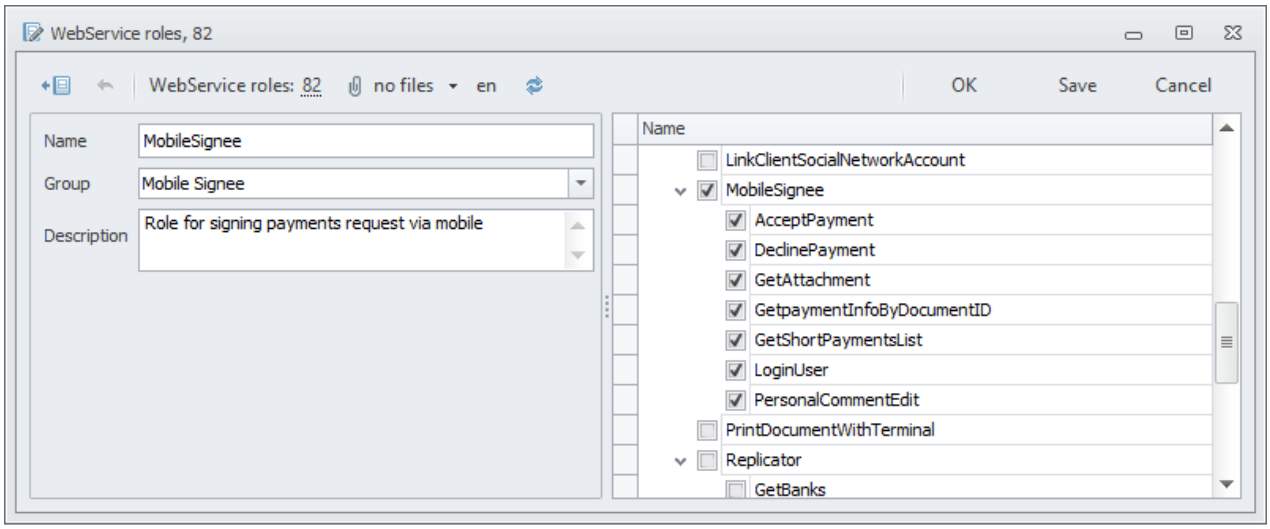
Tools of web service settings are detailed described in documentation of the system administrator Ultimate AEGIS®. Tools are in the *Administrator* tab of the main menu in *WebService* group:



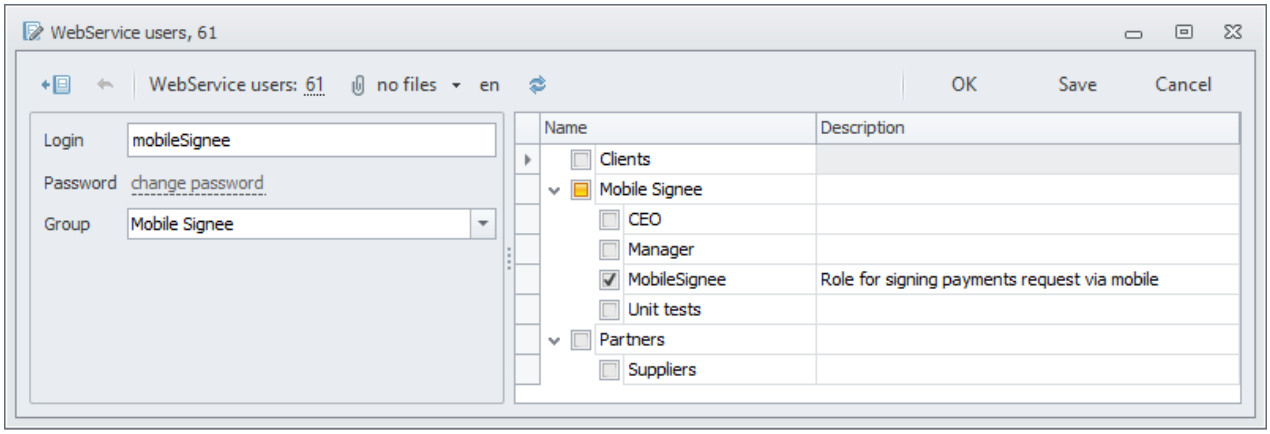
- *Users* – web service users;
- *Roles* – web service roles;
- *Sessions* – web service sessions.

It is necessary the following for web service settings:

- as *web service role* to set a flag in the web service list for this web service:



- in a *web service user* card to set a flag in a role list for the role giving access to this web service:



Several roles may be assigned to an user. The role, in its turn, can provide access to several web services. As a result, the user gets access to all web services of all selected models.

Pay attention: Accounts with unlimited mode of connection accounting are listed in a license file. If an account login is designated as unlimited, a password will be checked according to license file but not in the user dictionary!

Example of client implementation in C#:

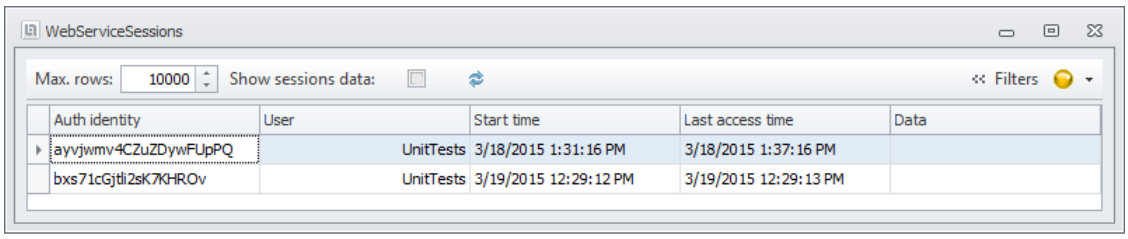
```
using System;
using ServiceStack;
using Ultima.WebServices;

namespace ConsoleApplication
{
	class Program
	{
		static void Main(string[] args)
		{
			var client = new JsonServiceClient("http://192.168.254.54:8337")
			{
				UserName = "UserName",
				Password = "UserPassword"
			};

			var request = new GetNow();
			var response = client.Get(request);

			Console.WriteLine("IsoTime: {0}", response.IsoTime);
			Console.WriteLine("Time: {0}", response.Time);
			Console.WriteLine("Timestamp: {0}", response.Timestamp);
			Console.WriteLine("Date: {0}", response.Date);

			Console.ReadLine();
		}
	}
}
```

Example of client implementation in php:

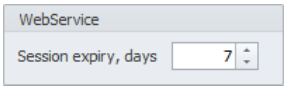
```
#!/usr/bin/php -q
<?php
require_once 'HTTP/Request2.php';
require_once 'HTTP/Request2/CookieJar.php';

$request = new HTTP_Request2("http://192.168.254.54:8337/json/reply/GetNow",
HTTP_Request2::METHOD_POST);
$request->setAuth('UserName', 'UserPassword');

$response = $request->send();

if ($response->getStatus() == 401)
{
	echo "Auth error!\n";
	exit();
}

if ($response->getStatus() != 200)
{
	echo "Status: " . $response->getStatus() . "\n";
	exit();
}

$body = $response->getBody();
echo $body . "\n";
?>
```

After authentication of web service user, a session, ID is created, which is returned by authentication method.

The list of all sessions of the users of web services can be found in the form *WebServiceSessions* (in the tab *Administrator* of the main menu in the group *WebService*):



The life span of the sessions is indicated in the cluster settings.



Implementation of web service methods has *UserSession* property with type *IWebServiceUserSession*. Using this property, you can learn the code of authorized user of the web service – *UserSession.UserID*, as well as handle the data stored for this session – *Dictionary<string, object>UserSession.Data*.

For example, a code of the client, who passed authentication procedure on the website, is stored in the base solution in *UserSession.Data*. If the application has its own system of sessions, its ID can be saved in the web service session.

In order to save the session between the calls of web service methods, cookie "ss-id" should be saved in the web service client and transferred.

The extended information about the errors is transferred to the web service client in HTTP headers *UltimaErrorCode*, *UltimaErrorText* and *UltimaUserReadableError*. The first header is error code, the second one is its text.

Codes 100-199 are reserved for the errors of basic solution. For example, the client's authentication error has code 100.

In order to bring back the extended information about the error, *ExtendedHttpError* class should be used:

```
throw new ExtendedHttpError(1234, "Error description", true);
```

Exception *ExtendedHttpError* is intended for use only and directly in the web code of services.

Use in other places, as a rule, means doubling of the logic – a situation in which procedures of data processing for web and not for web differ. It should be avoided carefully. While respecting the exception does not need to throw out anywhere except web services code.

For processing of expanded information on web service errors the clients of ServiceStack should receive data of http headings from the exception:

```
try
{
	var request = new GetNow();
	var response = client.Send(request);
}
catch (WebServiceException e)
{
	var errorCode = e.ResponseHeaders.GetValues("UltimaErrorCode");
	var errorText = e.ResponseHeaders.GetValues("UltimaErrorText");
}
```

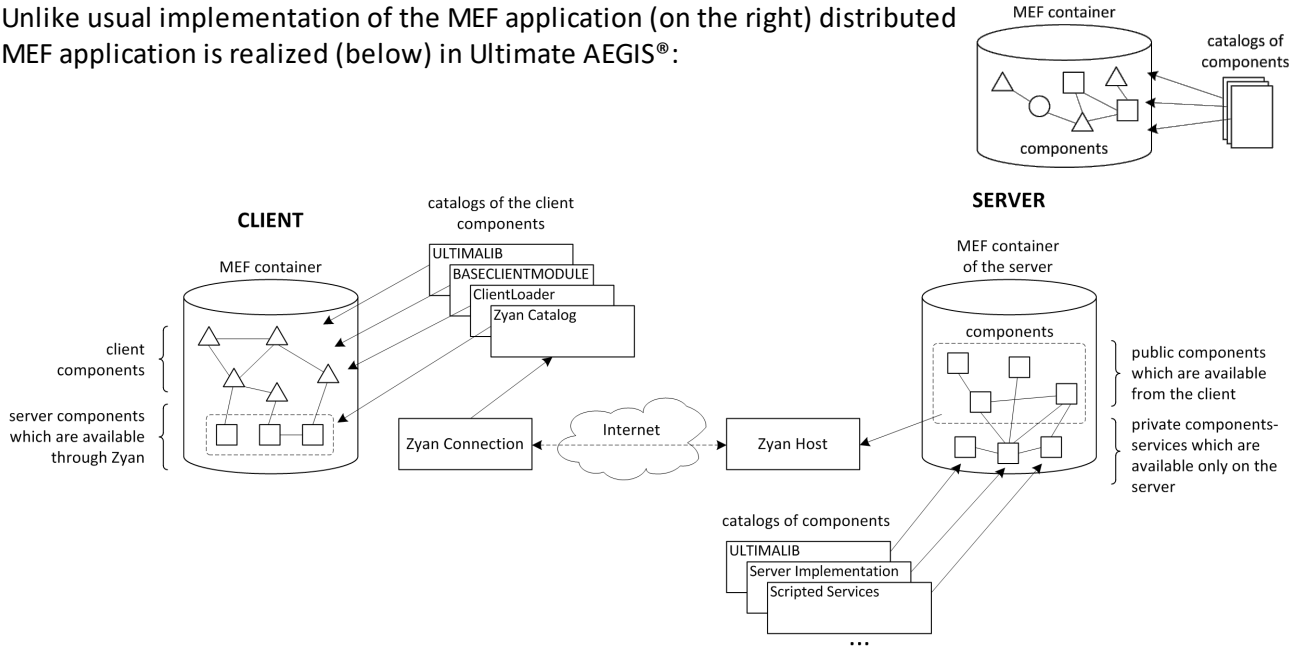
Use of services

To use the functionality provided by services, it is necessary to import the interface realized by them. For this purpose it is necessary to announce the property of the set type and to mark it with attribute [Import]:

```
[Import]
private IServiceName ServiceName { get; set; }
```

Import is carried out by means of the MEF platform (Managed Extensibility Framework) which detailed description can be found on MSDN website ⇨ eng/rus.

Unlike usual implementation of the MEF application (on the right) distributed MEF application is realized (below) in Ultimate AEGIS®:



To import multiple services that implement the same interface *ImportMany* is used (*ImportSource* must be specified, as a hierarchy MEF-containers is used in Ultimate AEGIS®):

```
[ImportMany(Source = ImportSource.Local)]
private Lazy<IServiceName>[] ServiceName { get; set; }
```

The inactive initialized list of services will be the result of such imports that implement the specified interface.

If it is necessary to choose one of them, it is possible to use import metadata:

```
[ImportMany(Source = ImportSource.Local)]
private Lazy<IServiceName, IServiceMetadata>[] ServiceName { get; set; }
```

MEF Explorer – debugging

The MEF platform (⇨ eng/rus) is the framework for the Ultimate AEGIS® system. It assembles an application from independent components, such as kernel services, scripts and applied classes. MEF is used both in server and client parts, making the program to have an identical modular structure in both cases and use single API, which is well documented and rather popular.

MEF is a late binding system based on the comparison rules. The comparison is carried out between so called imports and exports provided by the components. To get a component ready to go, all its imports must be compared with exports of other components. Binding is carried out during the execution of the program, which ensures the desired flexibility. The components to build the program may be developed

simultaneously by separate programmers' teams: e. g., the form of a client application may use a server service, which is under construction, provided that the service interface is formally coded.

A downside of such flexibility is binding errors. If in the process of execution the application needs a service that is not yet implemented, a binding error will occur. The program components may have sophisticated relations, and even if one such relation is not found, the whole component becomes useless.

Unfortunately, the diagnosis of such bugs is difficult. All that MEF knows at the moment of occurrence of a binding error is that one of the binding rules cannot be complied with. A typical text of the bug in such case is as follows:

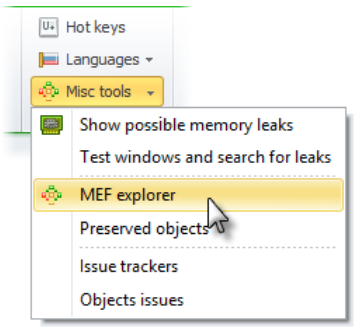
```
The composition produced a single composition error. The root cause is provided below.
Review the CompositionException. Errors property for more detailed information.

1) No exports were found that match the constraint:
ContractName          Ultima.Scripting.IUserCommand
RequiredTypeIdentity  Ultima.Scripting.IUserCommand
RequiredMetadata
    ScriptID          (Ultima.Scripting.IScriptMetadata)

Resulting in: Cannot set import '
    ContractName            Ultima.Scripting.IUserCommand
    RequiredTypeIdentity    Ultima.Scripting.IUserCommand
    RequiredMetadata
        ScriptID            (Ultima.Scripting.IScriptMetadata)' on part '(name)'

Element:
    ContractName            Ultima.Scripting.IUserCommand
    RequiredTypeIdentity    Ultima.Scripting.IUserCommand
    RequiredMetadata
        ScriptID            (Ultima.Scripting.IScriptMetadata) --> Unknown Origin

   at System.ComponentModel.Composition.CompositionResult.ThrowOnErrors(AtomicComposition
atomicComposition)
   at
System.ComponentModel.Composition.Hosting.ImportEngine.SatisfyImportsOnce(ComposablePart
part)
   at
System.ComponentModel.Composition.Hosting.CompositionContainer.SatisfyImportsOnce(Composab
lePart part)
```

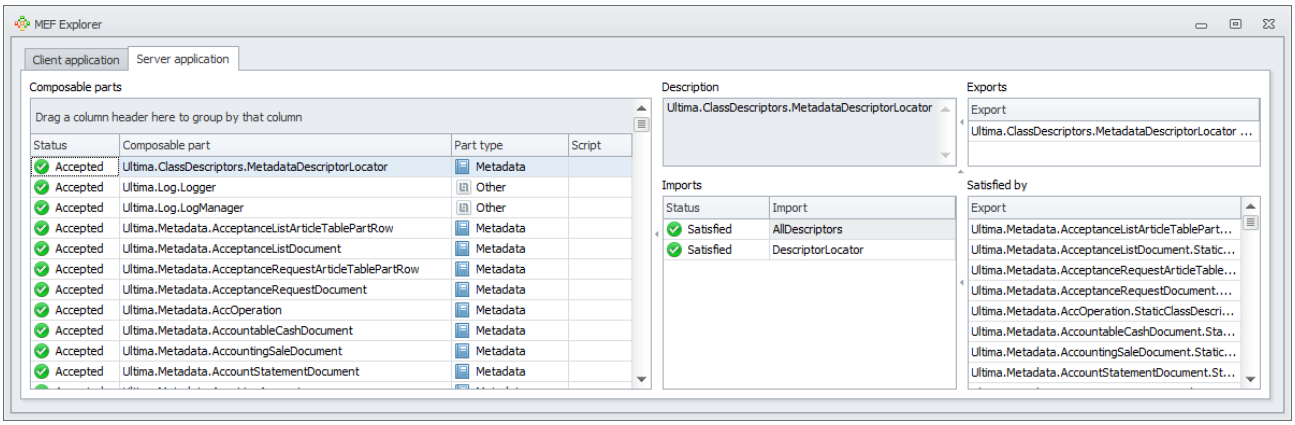
It is only clear from the text that some component cannot be provided upon the system request. Which component exactly is the cause of the error and how to correct it is unclear.

Since it is only applied errors of such kind that are of interest to programmers, the range of possible causes narrows down to two typical situations:

- error in a script (not necessarily in that one requested);
- error in a client form (or in one of its relations).

To investigate such errors on the basis of the console utility Mefx, a tool called *MEF explorer* was developed. It shows the structure of MEF catalogs for client and server parts (in the corresponding tabs).





If no errors occurred, all component imports are compared with exports of other components, and all the composable parts are supplied with the status *Accepted*.

Of course, other statuses are of greater interest. Any binding error means that the component was rejected due to one of the following reasons:

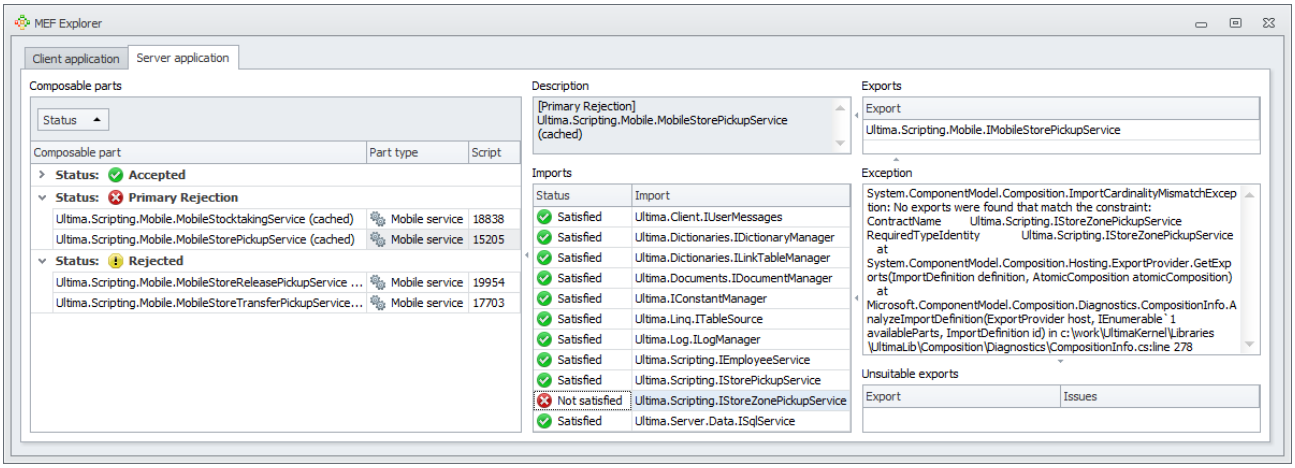
Primary Rejection – particular component's imports lack corresponding exports;

Rejected – component's exports exist, but cannot be created, as their imports, in their turn, lack corresponding exports.

Let's see, what actions the application programmer will perform, once the error occurred.

Suppose that the binding error (No exports were found that match the constraint...) occurs during the *MobileStorePickupService* request. However, the real problem is never in the request. If to open the script in an editing program, we will see that it is compiled without errors. To identify the reason, run *MEF Explorer* and search for the component that caused the error in the *Composable parts* list by its statuses, *Rejected* or *Primary Rejection*, and name.

When selecting this component in the *Imports* list, to the right of the component, there will be shown all imports that are either bound or unbound with exports. Viewing unbound imports, we can find out the real cause of the error:

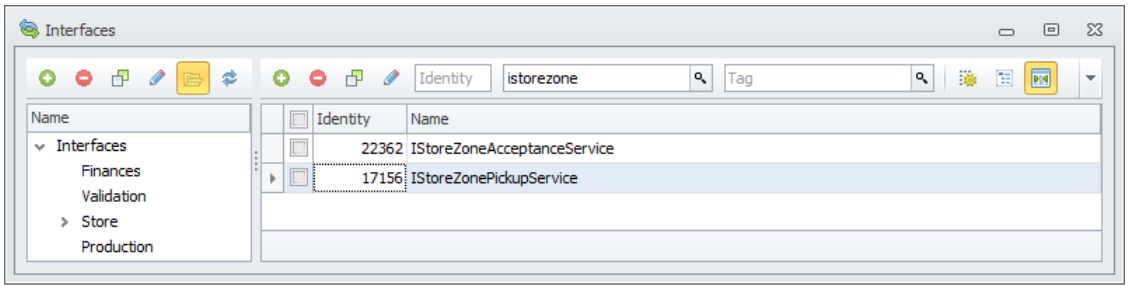


In the example above, we can see that the problem is in *IStoreZonePickupService*. This application service is absent in the server-side catalog. There may be the following reasons for that:

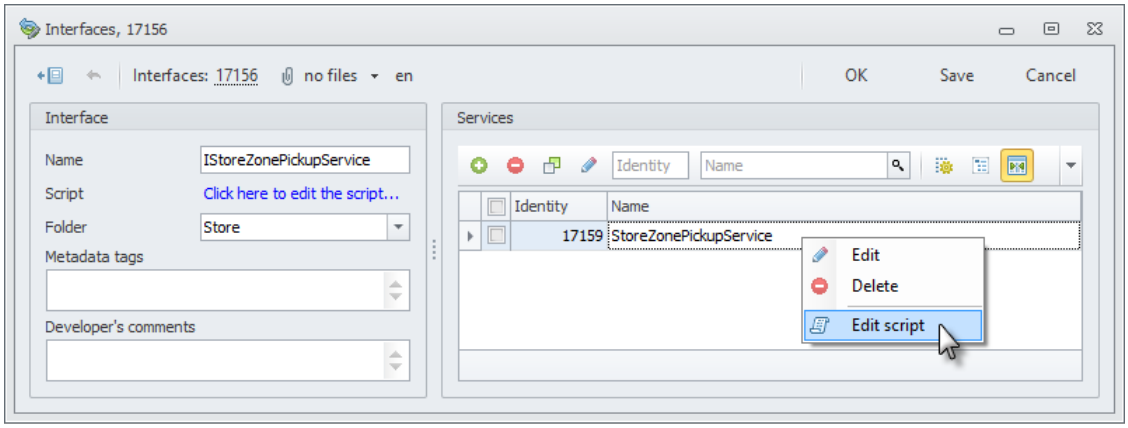
- the service has an interface, but it is not implemented so far;
- the interface is implemented, but there are compilation errors;
- the MEF Cache of the service script is empty for some reason (probably, as a result of merging of metadata versions).

To correct the error, you need to:

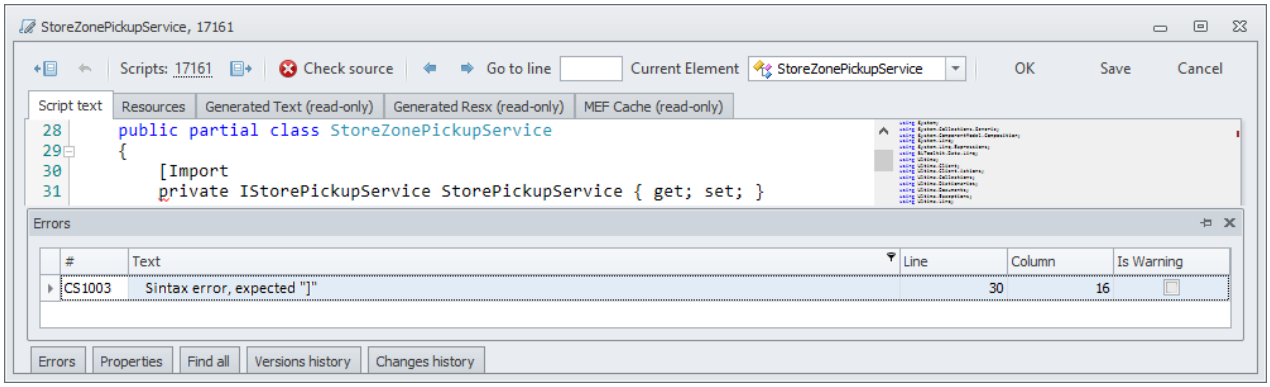
- find the *IStoreZonePickupService* interface in the *Interfaces* dictionary:



- check if the interface is implemented (if not, add implementation):



- if implemented, repair the implementation script:



Data access methods

Data access is possible using one of the following methods:

- services *IDictionaryManager* and *IDocumentManager*:
  - are designed to manage data (saving, creation, deletion) of system objects;
- LINQ queries allows carrying out:
  - strictly typed access to the objects;
  - JOIN operations and complex selection conditions;
- SQL service – allows executing:
  - ad hoc SQL queries (read only);
  - autonomous transactions;
  - query to Standby server;
  - sequencer;
  - handling of temporary tables (TEMP_TAG_IDLIST);

IDocumentManager and IDictionaryManager interfaces_2

The system provides *DictionaryManager* to manipulate the dictionary records.

The system provides *DocumentManager* to manipulate the documents.

The description of all other managers can be found in the section Special managers.

The manager (from namespace *Ultima.Dictionaries*) serves for work with dictionaies .

The interface of *IDictionaryManager* implements the following methods:

- *GetRecord (Type dictionaryType, long id, bool withInnerObjects = false)* returns an instance of dictionary record with specified ID:
  - *dictionaryType* – dictionary type;
  - *id* – dictionary record ID;
  - *withInnerObjects* – if the parameter value is set to *true*, the inner object of dictionary record will be loaded. The inner objects of dictionary record include e.g. link tables and attached dictionaries. In case of indication of the need in loading of inner objects, the data of link tables associated with the dictionary record and of attached dictionaries will be loaded in full. Otherwise, only IDs will be loaded;
- *GetRecords(Type dictionaryType, LambdaExpression selectExpression, IDList records)* returns a table of dictionary records with specified IDs:
  - *dictionaryType* – dictionary type;
  - *selectExpression* – an expression describing which of dictionary columns will be loaded. If null is indicated as parameter value – the values of all columns will be loaded for indicated dictionary records;
  - *IDList* – a list of IDs of dictionary records. If null is indicated as parameter value – all dictionary records will be loaded;
- *GetRecords(Type dictionaryType, LambdaExpression selectExpression, LambdaExpression filterExpression)* returns a table of dictionary records meeting the filter condition:
  - *dictionaryType* – dictionary type;
  - *selectExpression* – an expression describing which of dictionary columns will be loaded;
  - *filterExpression* – an expression describing which of dictionary records will be loaded (see details in the section Filters);
- *GetLookup(Type dictionaryType, IDList records = null)* returns a table of dictionary records with specified IDs containing only lookup-columns (properties of the dictionary with set flag *LookUp* and if not, it is listed in the *DisplayFormat* properties):
  - *dictionaryType* – dictionary type;
  - *IDList* – a list of IDs of dictionary records. If null is indicated as parameter value – all dictionary records will be loaded;
- *GetLookup(Type dictionaryType, LambdaExpression filterExpression)* returns a table of dictionary records meeting the filter condition, and containing only lookup-columns:
  - *dictionaryType* – dictionary type;
  - *filterExpression* – an expression describing which of dictionary records will be loaded (see details in the sectionFilters);
- *NewRecord(Type dictionaryType, IDictionary<string, object> parameters = null, IDictionaryRecord template = null)* creates a new dictionary record and transfers it to BeforeCreate handler:
  - *dictionaryType* – dictionary type;
  - Parameters— the parameter list for the new entry (optional);
    - as parameters, you can use any of the properties in the directory entry;
    - any additional parameters can be processed by the script dictionary.
    - if set to ID, this value will be used for new record ID;
  - *Template– initial value (optional);*

- *SaveRecord(Type dictionaryType, IDictionaryRecord record, params ILinkTable[] linkTables)* saves the dictionary record:
  - *dictionaryType* – dictionary type;
  - *record* – a dictionary record being saved;
  - *linkTables* – data of link tables for saving;
- *SaveRecords(Type dictionaryType, IDictionaryTable records)* saves the dictionary record:
  - *dictionaryType* – dictionary type;
  - *records* – a table of dictionary records being saved;
- *SaveRecords(params IEntity[] records)* saves the dictionary records:
  - *records* – dictionary records being saved;
- *SaveRecords(RecordSet recordSet)* saves the dictionary records:
  - *recordSet — a set (*container) save the phonebook entry;
- *SaveAndGetRecord(Type dictionaryType, IDictionaryRecord record)* saves the dictionary record and then retrieves it from the database, returns an instance of loaded record:
  - *dictionaryType* – dictionary type;
  - *record* – a dictionary record being saved;
- *DeleteRecord(Type dictionaryType, long id)* deletes the specified dictionary record:
  - *dictionaryType* – dictionary type;
  - *record* – ID of the dictionary record being deleted;
- *CloneRecord(Type dictionaryType, long id)* clones the specified dictionary record:
  - *dictionaryType* – dictionary type;
  - *record* – ID of the dictionary record being cloned.

```
[Import]
private IDictionaryManager DictionaryManager { get; set; }

var user = DictionaryManager.NewRecord<User>();
DictionaryManager.SaveRecord(user);
```

RecordSet class – container of records for saving:

- *AddTask(long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* adds a print task for the print form, not associated with any object:
  - *DictionaryRecords* – collection of dictionary records;
  - *LinkRecords* – collection of link table record;
  - *documents* – documents collection;
  - *DictionariesTables* – collection of dictionaries table;
  - *LinkTables* – link tables collection.

interface*IEntityExtensions* (from namespace *Ultima.EditableObjects*) realize the extend interface methods *IEntity* – of single methodsmetadata objects classes (dictionaries, documents, etc) - and let to receive original values of dictionary record properties:

- *GetOriginalValue<TEntity, TResult>(this TEntity entity, Expression<Func<TEntity, TResult>> getExpr)* – backs original value of stated property:
  - *TEntity* – object type;
  - *TResult* – result type;
  - *entity* – object;
  - *getExpr* – expression of receiving the property of type *x => x.Name*;
- *IsPropertyChanged<TEntity, TResult>(this TEntity entity, Expression<Func<TEntity, TResult>> getExpr)* – backs*true*, if the value of stated property was changed:
  - *TEntity* – object type;
  - *TResult* – result type;
  - *entity* – object;
  - *getExpr* – expression of receiving the property of type *x => x.Name*;

- *RejectPropertyChanges<TEntity, TResult>(this TEntity entity, Expression<Func<TEntity, TResult>> getExpr)* – backs original value to the stated property:
  - *TEntity* – object type;
  - *TResult* – result type;
  - *entity* – object;
  - *getExpr* – expression of receiving the property of type *x => x.Name*;
- *SetPropertyValues(this IEntity entity, IDictionary<string, object> propertyValues)* – let entry values of scalar property:
  - *entity* – object;
  - *propertyValues* – values set.

```
long userId = 1;
var user = DictionaryManager.GetRecord<User>(userId);
user.Name = user.Name + "Junior";

var currentValue = user.Name; // current (changed) value of the property
var  originalValue = user.GetOriginalValue(x => x.Name); // original value
user.IsPropertyChanged(x => x.Name); // return true
user.RejectPropertyChanges(x => x.Name); // return original value to the property
```

The manager (from *Ultima.Documents* namespace) is designed to handle documents.

The interface of *IDocumentManager* implements the following methods:

- *GetDocumentType(long id, bool demandReadPermission = false)* returns a type of specified document and, optionally, if current user has a permission to read it:
  - *id* – document ID;
  - *demandReadPermission* – if *true* it checks where current user has a permission to read the document;
- *GetDocumentTypeID(long id)* returns ID of specified document:
  - *id* – document ID;
- *GetDocumentSubtypeID(long id)* returns subtype ID of specified document:
  - *id* – document ID;
- *GetDocument(long id, bool includeDeleted = false)* returns an instance of the document with specified ID:
  - *id* – document ID;
  - *includeDeleted* – if *true* is indicated as parameter value, the values will be returned also for deleted rows of the table parts of the document (but only for those table parts, for which *Soft deletion* is enabled);
- *NewDocument(Type documentType, long? subtypeId = null, IDictionary<string, object> parameters = null, IDocument template = null)* – creates a new document, returns an instance of created document of certain subtype (if the subtype is not defined, an exception will be returned):
  - *documentType* – document type;
  - *subtypeId* – document subtype (optional);
  - *Parameters- a list of parameters to create (optional);*
    - as parameters it is possible to use any properties of a header of the document;
    - any additional parameters can be processed by a document script;
    - if ID is set in parameters, then this value will be used to code a new document;
  - *Template – initial value (optional);*
- *SaveDocument(IDocument document)* saves the document:
  - *document* – a document being saved;
- *SaveDocuments(params IDocument[] documents)* saves the documents:
  - *document* – a document being saved;

The methods checks if each of the documents is delivered exactly once. This prevents from the situation, when the same document in a list is in two different states, such as:

```
var doc1 = DocumentManager.GetDocument<SaleDocument>(123);
var doc2 = DocumentManager.GetDocument<SaleDocument>(123); // the same document code
doc1.AgentID = 1;
doc2.AgentID = 2; // different fields values
SaveDocuments(doc1, doc2); // document state is undefined
```

- *SaveAndGetDocument(IDocument document)* – saves the document, receives it from the database and then returns the copy of the loaded document:
  - *document* – a document being saved;
- *DeleteDocument(long id)* – deletes the specified document:
  - *id* – ID of the document being deleted;
- *ReviveDocument(long id)* – tries to revive the specified document:
  - *id* – ID of the document being revived;
- *ReprocessDocuments (IDList documents) - to recreate the wiring of these documents, without changing anything and without going through the event handlers:*
  - *documents* – a list of documents IDs;
- *AddLink(long parentDocumentId, long childDocumentId, long linkTypeId)* – created link between two documents:
  - *parentDocumentId* – parent document ID;
  - *childDocumentId* – child document ID;
  - *linkTypeId* – link type ID;
- *RemoveLink(long parentDocumentId, long childDocumentId, long? linkTypeId = null) - removes the link between the two documents:*
  - *parentDocumentId* – parent document ID;
  - *childDocumentId* – child document ID;
  - *linkTypeId— a reference code which needs to be removed (if it isn't specified — the reference of any type will be removed);*
- *RemoveLinks(long documentId, long? linkTypeId = null) - removes the link between the two documents:*
  - *documentId – document ID;*
  - *linkTypeId— a reference code which needs to be removed (if it isn't specified — the reference of any type will be removed);*
- *GetDocumentParents(long documentId)*— returns a list of IDs of parent-documents of the specified document:
  - *documentId* – document ID;
- *GetDocumentChildren(long documentId)*— returns a list of IDs of child-documents of the specified document:
  - *documentId* – document ID;
- *GetDocumentFamily(long documentId)*— returns a list of IDs of family-documents of the specified document:
  - *documentId* – document ID;
- *GetAllowedSubtypes(long typeId, AccessOperation accOperation) — returns a list of subtypes for specified document type, the access to execution of specified operation over which the current user has:*
  - *typeId* – document type ID;
  - *accOperation* – operation.

```
[Import]
private IDocumentManager DocumentManager { get; set; }

var newDoc = DocumentManager.NewDocument<SaleDocument>();

var oldDoc = DocumentManager.GetDocument<PurchaseDocument>(123);

// Creation of document copy with the table parts.
// The flags Save and Deleted are not copied in such case.
var copyDoc = DocumentManager.NewDocument(oldDoc);
DocumentManager.SaveDocument(copyDoc);
```

interface IEntityExtensions (from namespace *Ultima.EditableObjects)* realize the extend interface methods *IEntity – of single methods* metadata objects classes (dictionaries, documents, etc) - and let to receive original values of dictionary record properties:

LINQ queries

An access to each object of the system can be gotten through the interface *ITableSource* (from namespace *Ultima.Linq*):

```
[Import]
private ITableSourse DataContext { get; set; }
```

The result of the LINQ inquiries will be collections of the type *IEnumerable<T>* (where *T* – collection element type):

```
var query =
	from u in DataContext.GetTable<User>()
	where u.Name.StartsWith("a")
	select new User
	{
		ID = u.ID,
		Name = u.Name,
		Login = u.Login
	};
```

Using LINQ inquiries it is also possible to get a collection of objects of anonymous classes:

```
var query =
	from u in DataContext.GetTable<User>()
	where u.Name.StartsWith("a")
	select new
	{
		u.ID,
		UserName = u.Name,
		u.Login,
		FullName = u.Name + " " + u.LastName
	};
```

And also to carry out the operations JOIN and to apply difficult conditions of selection:

```
var branches =
	from ver in DataContext.GetTable<VersionTreeLink>()
	join tag in DataContext.GetTable<VersionTag>()
		on ver.DescendantID equals tag.VersionID
	where tag.IsBranch && ver.AncestorID == rootVersionId
	select tag;
```

To carry out an asynchronous loading of data:

```
var result = await DataContext.GetTable<User>().ToListAsync();
```

The following methods of asynchronous loading of data are also available: *SingleAsync()*, *SingleOrDefaultAsync()*, *FirstAsync()*, *FirstOrDefaultAsync()*, *ToArrayAsync()*, *ToDictionaryAsync(), ToIDListAsync(), SingleAsync(), SingleOrDefaultAsync(), FirstAsync(), FirstOrDefaultAsync(), AnyAsync(), AllAsync()*.

Besides, it is possible to use the filter, constructed by means of PredicateBuilder, in LINQ inquiries.

SqlService

SqlService (from *Ultima.Server.Data* namespace) allows executing ad hoc SQL queries:

```
// A query without parameters.
var sql = @"SELECT DBMS_RANDOM.RANDOM
       FROM DUAL CONNECT BY ROWNUM <= 10";

var randomTable = SqlService.Select(sql);

// Query with parameters.
var sql = @"
       SELECT * FROM USER_OBJECTS
       WHERE OBJECT_TYPE = :vType AND STATUS = :vStatus";

var parameters = new Dictionary<string, object>
{
       { "vType", "INDEX" },
       { "vStatus", "VALID" }
};

var indexes = SqlService.Select(sql, parameters);
```

Use of *SqlService* is possible in a separate transaction within one server call. In the example below, the entire block inside `using` will be registered in the base as single operation, irrespective of the server call results:

```
using (var scope = SqlService.BeginTransaction())
{
       var args = new Dictionary<string, object>
       {
              { "vID", id }
       };

       var sql = @"DELETE FROM MY_TEST_TABLE WHERE ID = :vID";
       SqlService.Execute(sql, args);
       scope.Complete();
}
```

If a circuit with standby server of DBMS is used, a part of load on it can be redistributed. The standby servers are open only for reading:

```
using (var db = SqlService.Standby())
{
       // execute read-only query
}
```

In some cases, the use of *SqlService* is the only possible method. For instance, new value of the sequence cannot be obtained using LINQ queries:

```
var barcodes = SqlService.GetNewID("STICKER_BARCODE_SEQ");
```

Besides, *SqlService* allows handling temporary tables:

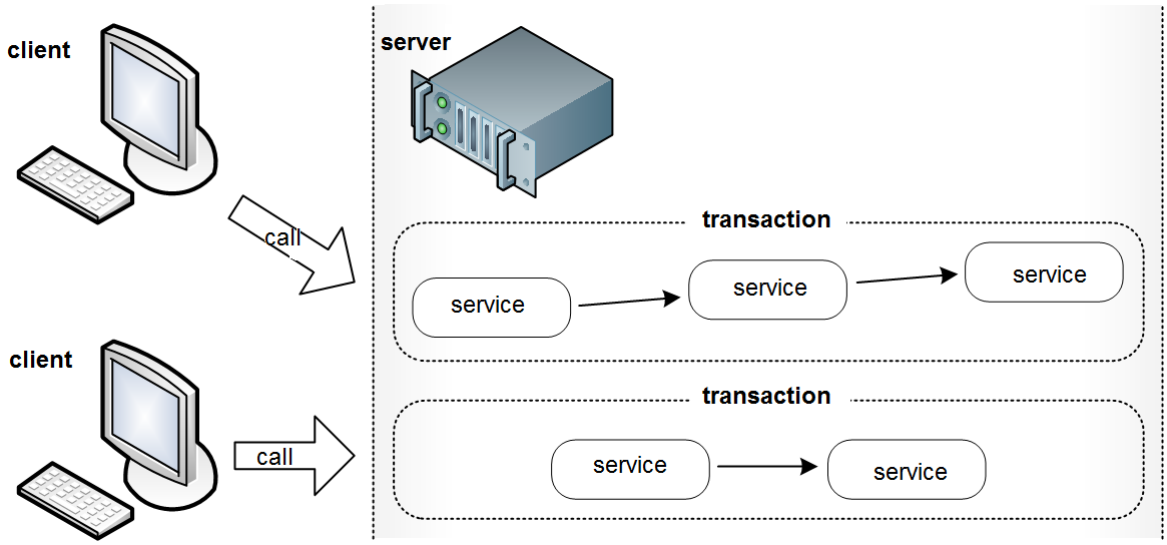
```
var tagId = SqlService.FillTempIDList(articleList);
string sql = @"
       SELECT TR.ARTICLE_ID, TR.CCD_ID
       FROM TR_STOCK_CCDS TR
       JOIN KERNEL.TEMP_TAG_IDLIST T
       ON T.ID = TR.ARTICLE_ID and T.TAG = :vTagID
var resultTable = SqlService.Select(sql);
```

SqlService implements the following methods:

- *Select(string sql)* executes SQL query without parameters, returns an object of *SlimTable* class:
  - *sql* – SQL query;
- *Select(string sql, IDictionary<string, object> parameters)* executes SQL query with the set parameters, returns an object of *SlimTable* class:
  - *sql* – SQL query;
  - *parameters* – a list of parameters for query execution;
- a family of *Execute* methods, executing SQL query:
  - *Execute(string sql)* executes SQL query without parameters, returns a number of records satisfying the query:
    - *sql* – SQL query;
  - *Execute(string sql, IDictionary<string, object> parameters)* executes SQL query with the set parameters, returns a number of records satisfying the query:
    - *sql* – SQL query;
    - *parameters* – a list of parameters for query execution;
  - *ExecuteScalar* returns an object, which is in the first column of the first row in the resulting query set;
  - *ExecuteScalar<T>* – strictly typed version of previous method;
  - *ExecuteList<T>* – strictly typed method, which returns the first column of the resulting query set;
- *GetReader(string sql)* executes SQL query without parameters, returns an object of *IDataReader* class:
  - *sql* – SQL query;
- *GetReader(string sql, IDictionary<string, object> parameters)* executes SQL query without parameters, returns an object of *IDataReader* class:
  - *sql* – SQL query;
  - *parameters* – a list of parameters for query execution;
- *GetNewID(string sequenceName)* returns a new ID for specified sequence (sequence):
  - *sequenceName* – sequence name;
- *GetNewIDs(string sequenceName, int count)* returns a set number of IDs for specified sequence (sequence):
  - *sequenceName* – sequence name;
  - *count* – ID count;
- *GetDatabaseTime()* – returns the database server's current timestamp;
- *FillTempIDList(IDList values)* fills in TEMP_TAG_IDLIST table with ID values. For instance, a price list should be generated for a large list of products meeting a number of conditions of complicated query: to obtain data from several tables – product name, its price, remaining stock, etc. In order not to execute the query for selection of products during execution of JOIN operation each time, it can be saved in temporary table. As a result of execution, the method returns a tag, which is created automatically and which a particular list of values in the temporary table is associated with:
  - *values* – a list of IDs for the record;
- *FillTempIDList(long tag, IDList values)* fills in a temporary TEMP_TAG_IDLIST table with ID values. It returns a tag, which is indicated also as parameter (the tag must be unique within the executed transaction) and which a particular list of values is associated with:
  - *tag* – tag;
  - *values* – a list of IDs for the record;
- *ExclusiveLock(string guid, string description = null)* gains exclusive lock:
  - *guid* – a unique ID of the lock;
  - *description* – description of the lock (optional);
- *BeginTransaction()* begins a new autonomous transaction, returns an instance of *TransactionScope*;
- *Standby()* connects to one of standby servers described in the cluster configuration. The standby servers are open only for reading: the data cannot be changed at the produced connection, even the temporary table cannot be filled in. It returns an object of *IDisposable* class.

Opening of new transaction

The main flow of server call is always executed in one transaction from start to end.



Exclusion at any stage of request processing will result into rollback of the entire transaction. However, execution of DDL requests (e.g. CREATE, ALTER, DROP) will commit the transaction, which may result into unstable behaviour of the system. Therefore, such requests (DDL or the requests, requiring intermediate registration due to other reason) should be executed in a separate transaction.

The situation may also arise, when within the scope of one server call any operation should be committed. For instance, during the request execution, changing a large number of rows (recalculation of prices), for reduction of duration of possible locks, it can be split into several parts, while changing 100 rows at once, and executing each part in a separate transaction, having commited it.

In order to open a new transaction within the scope of server call, class *TransactionScopeBuilder* should be used (from namespace *Ultima.Server.Data*):

```
// Clearing of table contents.
using (var scope = TransactionScopeBuilder.CreateTransactionScope())
{
	var sql = @"DELETE TABLE TEMP_CALCULATIONS";
	SqlService.Execute(sql);
	scope.Complete();
}
```

In the provided example, the entire block inside `using` will be registered in the base as single operation, irrespective of the server call results.

The use of *Complete* method indicates that all operations within the scope are completed successfully. The use of *Dispose* method ends the transaction scope, and if *Complete* method was not used before that, all operations in the scope will roll back.

The level of isolation for the transaction created using *TransactionScopeBuilder* will be *read commited*. If the transaction should be created with the level of isolation *serializable*, class *TransactionScope* can be used (detailed description of the class can be found on MSDN website ⇨ eng/rus):

```
using (var ts = new TransactionScope(TransactionScopeOption.RequiresNew))
```

However, the first option with *TransactionScopeBuilder* is recommended one.

Parallel execute request

There are situations in case of which several requests can be executed in parallel. It will allow reducing overall operation runtime to runtime of the most long executed request and as a result duration of possible locks.

The server allows starting flows in one server call. At the same time it is necessary to control completion of flows independently. In each created flow separate connection with DBMS will be used. It is necessary to consider that changes made earlier in the server call won't be visible from this new connection.

The *ServerCall* class (from *Ultima.Server.SessionMgmtnamespace*) has *RunParallel method* which accepts an array of objects like *Action* or *Function* and each of them will be started in a separate flow. The method itself will check completion of all flows.

The flow can be created by any other convenient method, at the same time it is necessary to create the *ServerCall class object independently*:

```
Task.Factory.StartNew(() =>
{
	using (new ServerCall())
	{
		//Operation execution.
	}
});
```

ServerCall.RunParallel starts tasks in parallel flows only if *ServerCall.AllowRunParallel = true*.

ServerCall class has four *RunParallel* methods:

- RunParallel(IEnumerable<Action> actions) executes an array of objects like *Action* in parallel flows:
  - *actions* – an array of objects like *Action* for execution;
- RunParallel<T>(IEnumerable<T> items, Action<T> action) executes operation (an object like *Action*) over an array of elements in parallel flows;
  - *T* – element type;
  - *items* – an array of elements;
  - *action* – an object like *Action* for execution at each of elements;;
- IEnumerable<R> RunParallel<R>(IEnumerable<Func<R>> functions) executes an array of objects like *Function* in parallel flows:
  - *R* – result type;
  - *functions* – an array of objects like *Function* for execution;
- IEnumerable<R> RunParallel<T, R>(IEnumerable<T> items, Func<T, R> func) – executes operation (an object like *Function*) over an array of elements in parallel flows;
  - *T* – element type;
  - *R* – result type;
  - *items* – an array of elements;
  - *func* – an object like *Function* for execution at each of elements.

All services (special managers and also kernel services and application services) are not threadsafe. If multithreaded programming is used on the server, each flow shall access to own copies of services:

```
[Import]
private IDictionaryManager DictionaryManager { get; set; }

private void BadCode(IDList users)
{
	//To write such code is invalid because several
	//flows address to one copy of DictionaryManager
	ServerCall.RunParallel(users, id =>
	{
		var user = DictionaryManager.GetRecord<User>(id);
		user.Name += " (Dismissed)";
		DictionaryManager.SaveRecord(user);
	});
}
```

Instead it is necessary to import the manager as follows:

```
[Import]
private ServerCallImport<IDictionaryManager> DictionaryManagerImport { get; set; }

private IDictionaryManager DictionaryManager
{
	get { return DictionaryManagerImport.Value; }
}
```

Multithreaded programming on the server belongs to opportunities which should be used only in case of emergency. Error connected to multithreaded execution is too easily to be made, and it is extremely difficult to find it. Potential advantage from use of the multithreaded (well, optimization of productivity of command execution) is often incommensurable to the error value (incorrect operation of the command after optimization).

Filters

The class *PredicateBuilder* is used (from namespace *Ultima.Linq*) to build expressions for the filters. Strictly typed expressions-predicates, which use extension-methods And or Or for combination of subexpressions, can be built using this class.

The class *PredicateBuilder* implements the following methods:

- PredicateBuilder.Get<T>(expr) returns a predicate *expr* for class *T*;
- PredicateBuilder.Get<T>() returns empty predicate (*null*) for class *T*;
- PredicateBuilder.True<T>() returns the predicate, which returns always *true* for class *T*;
- PredicateBuilder.False<T>() returns the predicate, which returns always *false* for class *T*;
- expr1.Or(expr2) – combines predicates *expr1* and *expr2* by means of *Or*, a result of such combination: *expr1 Or expr2;*
- expr1.And(expr2) – combines predicates *expr1 and expr2* by means of *And*, a result of such combination: *expr1 And expr2*.

Let us consider building the filter using *PredicateBuilder* by example:

Let it be a form containing two entry fields for adding: First Name and last Name. Tasks: to build an expression-filter for class Employee, using the data from these input fields in combination Or (i.e. the filter will ignore all records, which first and last names contain entered data). If any of input field is empty, a corresponding part of the filter will not be used. Therefore, if both fields are empty, the filter of records must appear empty too:

```
var filter = PredicateBuilder.Get<Employee>();

	if (!string.IsNullOrWhiteSpace(firstName))
	{
		filter = filter.Or(e => e.FirstName.Contains(firstName));
	}

	if (!string.IsNullOrWhiteSpace(lastName))
	{
		filter = filter.Or(e => e.LastName.Contains(lastName));
	}
```

If the data are entered into both fields, the result of PredicateBuilder functioning will appear the following expression:

```
e => e.FirstName.Contains(firstName) || e.LastName.Contains(lastName);
```

If only firstName is entered, just the first part of the filter will remain:

```
e => e.FirstName.Contains(firstName);
```

If both fields are empty, the formed filter will remain empty too (null).

Special managers

There are following special managers in the system:

- *IAuthManager* – manager of work with service of authorization and authentication (from namespace *Ultima.Auth*);
- *ICalendarManager* – manager of work with a calendar and the scheduler (from namespace *Ultima.Client*);
- *IClusterService* — provides cluster-related services such as broadcasting events to multiple servers (from namespace *Ultima.Server*);
- *IConstantManager* – manager of work with constants(from namespace *Ultima*);
- *IDictionaryCommandManager* – manager of work with dictionary list commands (from namespace *Ultima.Scripting*);
- *IDictionaryListCommandManager* – manager of work with dictionary list commands (from namespace *Ultima.Scripting*);
- *IDictionaryManager* – manager of work with dictionaries (from namespace *Ultima.Dictionaries*);
- *IDocumentCommandManager* – manager of work with document commands (from namespace *Ultima.Scripting*);
- *IDocumentListCommandManager* – manager of work with document list command (from namespace *Ultima.Scripting*);
- *IDocumentManager* – manager of work with documents (from namespace *Ultima.Documents*);
- *IEmailService* – manager of work with service of sending e-mail messages (from namespace *Ultima*);
- *IExportManager* – manager of work with service of export of printing forms (from namespace *Ultima*);
- *IHistoryService* – manager of management of logging in a DB (from namespace *Ultima*);
- *ILinkTableManager* – manager of work with link tables (from namespace *Ultima.Dictionaries*);
- *ILogger* and *ILogManager* – manager of logging (from namespace *Ultima.Log*);

- *INotificationService* – manager of work with service of sending notifications to the user (from namespace *Ultima.Client*);
- *IPrintManager* – manager of printing (from namespace *Ultima.Printing*);
- *ISmsService* – manager of work with service of sending short messages SMS (from namespace *Ultima*);
- *ITotalsManager* – manager of work with totals(from namespace *Ultima.Totals*);
- *IUserCommandManager* – manager of work with user commands (from namespace *Ultima.Scripting*);
- *IUserManager* – manager of work with users(from namespace *Ultima*);
- *IUserManager* – manager of work with user messages (from namespace *Ultima.Client*).

To use the functionality provided by managers, it is necessary to import their interfaces like this:

```
[Import]
private IDictionaryManager DictionaryManager { get; set; }
```

IAuthManager

The manager (from *Ultima.Auth* namespace) is designed for work with the authorization and authentication service. All actions in Ultimate AEGIS® system are performed under a particular user, who has a finite set of rights. The rights determine if the user can do a certain action or get access to a certain object. The ideology of this system is that any action or possibility of access to an object must be allowed initially, that is, what is not allowed is prohibited.

The system of user rights and its structure are described in details in *administrator's manual*, the section "Access control tools" (chapter "Administrator module functionality").

The *IAuthManager* interface has the following properties and implements the following methods:

- *IsAllowed(long permissionId)* – returns *true*, if the current user is granted access limited by the permission specified:
  - *permissionId* – permission ID;
- *IsAllowed(KernelPermissions permission)* – returns *true*, if the current user is granted access limited by the kernel permission specified (*KernelPermissions*):
  - *permission* – kernel permission (*KernelPermissions*);
- *UserHasPermission(long userId, long permissionId)* – returns *true*, if the current user is granted the permission specified:
  - *userId* – user ID;
  - *permissionId* – permission ID;
- *UserHasPermission(long userId, KernelPermissions permission)* – returns *true*, if the user specified is granted the kernel permission specified:
  - *userId* – user ID;
  - *permission* – *KernelPermissions*; can accept values:
    - *None* = 0 – no permissions;
    - *LoginAs* = 1 – allowed to sign in under;
- *UserHasDictionaryPermission(long userId, Type dictionaryType, AccessOperation accOperation)* – returns *true*, if the user specified is granted the specified permission to perform operations on the dictionary:
  - *userId* – user ID;
  - *dictionaryType* – dictionary type;
  - *accOperation* – *AccessOperation*; can accept values:
    - *Read* = 1;
    - *Create* = 2;
    - *Update* = 4;
    - *Delete* = 8;

- *UserHasSubTypePermission(long userId, long subTypeId, AccessOperation accOperation)* – returns *true*, if the user specified is granted specified permission to perform operations on the document's subtype:
  - *userId* – user ID;
  - *subTypeId* – document's subtype ID;
  - *accOperation* – *AccessOperation*;
- *IsAllowed(Type dictionaryType, AccessOperation accOperation)* – returns *true*, if the current user is granted the permission to perform the specified operations on the dictionary:
  - *dictionaryType* – dictionary type;
  - *accOperation* – *AccessOperation*;
- *IsAllowed(long subtypeId, AccessOperation accOperation)* – returns *true*, if the current user is granted the permission to perform the specified operations on the document's subtype:
  - *subTypeId* – document's subtype ID;
  - *accOperation* – *AccessOperation*;
- *IsScriptAllowed(long scriptId)* – defines if the script specified can be executed:
  - *scriptId* – script ID;
- *CurrentUserID* of *long* type – returns ID of the current user;
- *CurrentUser* of *UltimaIdentity* type – returns the current user;
- *DemandPermission(long permissionId, string operation = null)* – defines if the user has the specified permission; if doesn't, gives a regular exception of "Permission denied" type. You don't have permission {permissionId}{permission name} which is necessary to {operation}":
  - *permissionId* – permission ID;
  - *operation* – operation that requires the permission. The value of the parameter is used in the text of the exception given if the user has no permission.

The key difference between *HasPermission* and *IsAllowed* is that *HasPermission* checks if the user physically has the permission, while *IsAllowed* checks if the user may perform an action or get access limited by the permission specified. E. g., when in unsafe mode (used in commands), if a user does not possess the specified permission, the *HasPermission* method will return *false*; at the same time, for the same permission, the *IsAllowed* method will return *true*.

The *IAuthManagerServer* interface (from *Ultima.Auth* namespace) is available only on the application server and, in addition to the methods of *IAuthManager*, offers to install unsafe mode using the *SetUnrestrictedMode()* method:

```
using (AuthManager.SetUnrestrictedMode())
{
	DoSomethingRestricted();
}
```

As a consequence, all permission checks that are out of the application server will be performed in safe mode.

In the converse case, when it is needed to temporarily disable the unsafe mode in commands and other scripts, which are run with administrator privileges, one can use the *SetRestrictedMode()* method.

The *IAuthManagerServer* is available only on the application server and, in addition to the methods of *IAuthManager*, offers to install unsafe mode using the *SetUnrestrictedMode()* method:

Examples of use:

```
// get current user ID
long userId = AuthManager.CurrentUserID;

// check if user has administrator privilege
if(AuthManager.UserHasPermission(userId, Permission.Administrator))
{
	// perform operation requiring administrator privilege
```

```
}
```

ICalendarManager

The manager (from namespace *Ultima.Client*) is designed to work with the calendar and planner.

The interface of *ICalendarManager* implements the following methods:

- *GetStatuses* – returns the dictionary of all statuses of the calendar in a format: identifier and tuple name/color corresponding to it;
- *AddStatus(string name, int color)* adds new status with *name* and *color*;
- *RemoveStatus(long id)* removes the status with *id*;
- GetDayStatuses(DateTime date) returns a list of statuses (IDs) for *date*;
- GetDayStatuses(List<DateTime> dates) returns a list of statuses (IDs) for the list of *dates*;
- *SetDayStatus(DateTime date, long statusId)* sets *statusId* status for the *date*;
- *RemoveDayStatus(DateTime date, long statusId)* removes *statusId* status for the *date*.

IClusterService

The service (from *Ultima.Server* namespace) provides cluster-related methods, such as broadcasting events across multiple servers.

IClusterService interface includes the following methods:

- *LoadClusterConfiguration()* — loads the cluster configuration. This method is used by the infrastructure and is called on server startup.
- *RefreshClusterLink(bool cascade)* — establishes the network connection with neighbor servers . This method is used by the infrastructure and is called on server startup:
  - *cascade* – indicates that neighbor servers should also call *RefreshClusterLink* for their neighbors;
- *Subscribe(string eventName, Delegate handler)* – subscribes to the cluster event with the given name:
  - *eventName* – the name of the event, including the class name, for example, *Ultima.Scripting.MyService.MyEvent*;
  - *handler* – event handler called when an event is raised;
- *Unsubscribe(string eventName, Delegate handler)* – unsubscribes from the cluster event with the given name:
  - *eventName* – the name of the event, including the class name, for example, *Ultima.Scripting.MyService.MyEvent*;
  - *handler* – event handler called when an event is raised;
- *Broadcast(string eventName, bool cascade, EventArgs args)* – raises the cluster event with the given name:
  - *eventName* – the name of the event, including the class name, for example, *Ultima.Scripting.MyService.MyEvent*;
  - *cascade* – indicates that neighbor servers should also call *Broadcast* method for their neighbors;
  - *args* – event arguments class (should be serializable).
- *Broadcast(string eventName, EventArgs args)* – raises the cluster event with the given name, without the cascading option:
  - *eventName* – the name of the event, including the class name, for example, *Ultima.Scripting.MyService.MyEvent*;
  - *args* – event arguments class (should be serializable).

- *GetCurrentCluster()* – returns the *AppCluster* dictionary record, representing the current application cluster.
- *GetCurrentAppServer()* – returns the *AppServer* dictionary record, representing the current application server.
- *GetCurrentConfig()* – returns the *ClusterConfiguration* dictionary record, representing the current application cluster configuration.
- *GetPlatformFeatures()* – returns the *PlatformFeatures* instance, describing the current platform features.
- *RestartCurrentAppServer()* – restarts the current application server.
- *RestartCurrentCluster()* – restarts the current application cluster.

IClusterServiceExtensions — extension methods for working with cluster events

Subscribe, *Unsubscribe* and *Broadcast* methods take event name as a string, which is error-prone due to lack of the compile-time checking. IClusterServiceExtensions class provides the same set of methods, i.e. *Subscribe*, *Unsubscribe* and *Broadcast*, that take expressions referencing the events instead of plain event names. These methods are recommended to be used instead of the original *IClusterService* methods because they automatically address the typos or event renaming issues.

- *Subscribe(Expression<Func<TSource, Delegate>> eventExpression, Delegate handler)* – subscribes to the given cluster event:
  - *eventExpression* – expression that references the event property, for example, *x => x.MyEvent*;
  - *handler* – event handler called when an event is raised;
- *Unsubscribe(Expression<Func<TSource, Delegate>> eventExpression, Delegate handler)* – unsubscribes from the given cluster event:
  - *eventExpression* – expression that references the event property, for example, *x => x.MyEvent*;
  - *handler* – event handler called when an event is raised;
- *Broadcast(Expression<Func<TSource, Delegate>> eventExpression, bool cascade, EventArgs args)* – raises the given cluster event:
  - *eventExpression* – expression that references the event property, for example, *x => x.MyEvent*;
  - *cascade* – indicates that neighbor servers should also call *Broadcast* method for their neighbors;
  - *args* – event arguments class (should be serializable).
- *Broadcast(Expression<Func<TSource, Delegate>> eventExpression, EventArgs args)* – raises the given cluster event, without the cascading option:
  - *eventExpression* – expression that references the event property, for example, *x => x.MyEvent*;
  - *args* – event arguments class (should be serializable).

IConstantManager

The manager (from namespace *Ultima*) is designed to handle users. Constants are meant for storing variables used in the software code. Although this type of object the name of the constants is selected and their value can be changed. Constants - versioned objects therefore can have different values on different branches. It is not necessary to use constants for preservation of intermediate results of calculations of scripts. Try to use constants reasonably — for example, for storage of seldom changing values of dictionary codes.

Bad examples of use of constants:

- Shop1Phone, Shop2Phone — storage of phones of shops in constants is unreasonable, you need to transfer this value directly to the dictionary of shops.
- ShippingTypeAvia, ShippingTypeRails — constant of this type is better to convert to constant dictionaries (see Dictionary, Other constants)

Good examples of use of constants:

- ArticlePriceRecalculationThreadCount
- BingKey

Manager interface has the following properties:

- *this[string name]* – returns or sets value of a constant with the name *name*.

To view existing and to create new constants it is possible in the dictionary Constants, which is described in detail in *administrator's guide* in the chapter "Functional Administrator module".

The system also generates strictly typified class-wrapper for constants for use in scripts:

```
var fromEmail = Constants.ChangePasswordFromEmail;
```

For appeal to constants, the special class, generated on all constants in the current branch of metadata, is used in scripts. Constants are presented by the typified properties in this class, therefore autofilling is available to the list of constants in the editor of the script text. New constants are available after compilation and metadata reset.

Class-wrapper is automatically added to all of the generated scripts as Constants property. To use IConstantManager in scripts is not recommended, as in this case the checking of the name and type of the constant is lost by the compiler.

Examples of use of constants in a script:

```
// correct
var prefix = Constants.ArticleBarcodePrefix;

// right
var userList = Constants.WebServiceUsers;

// incorrect!
var maxReserveAmount = (decimal)ConstantManager["MaxReserveAmount"];
```

Wrapper-Class looks like

Strongly typed wrapper for constant looks like a class with properties, names and types that duplicate names and types of constants. Values of constants are not threaded in a class body, and requested dynamically from service IConstantManager. At addition of a new constant, renaming or change of the type recompilation of metadata is required, at value change — it is not required.

The generated class looks like this:

```
public class RootConstantGroup
{
     private IConstantManager ConstantManager { get; set; }

     public long DefaultFirmID { get { return (long)ConstantManager["DefaultFirmID"]; } }

     public decimal MaxReserve { get { return (decimal)ConstantManager["MaxReserve"]; } }
     ...
}
```

Constants of documents subtypes

The constants, which can be used in the scripts and application modules, are generated automatically for the subtypes of documents.

The names of constants correspond to the names (*Name*) of subtypes, which are set in the tab "Subtypes" of document type. In case of saving of the document type, the constants will be generated for all of its subtypes.

Advantages of use of named constants:
- readability (it can be seen immediately what the type it is);
- impossibility to indicate existing subtype or subtype of another's document type.

Example of use:

```
doc.SubtypeID = ReserveDocument.Subtypes.Reserve;
```

Constants become available to use in scripts after compilation and updating of assembly of metadata.

Dictionary constants

For dictionaries the constants are generated, specified in the list of constants in the form of editing of the dictionary. These constants are compiled together with metadata of dictionaries and put into the code together with their values. It makes sense to create constants of dictionaries in small dictionaries-listings: Types of agents, Currencies, Languages and so on. Constants are usually not necessary in dictionaries like Agents or Goods.

Use these constants when values of constants in principle can not be changed. Examples of suitable constants of dictionaries:

- Language.Constants.English — an English code language
- Currency.Constants.Ruble — a currency code is Ruble
- Barcode.Constants.Empty — a code for an empty barcode.

Example of use:

```
doc.CurrencyID = Currency.Constants.Ruble;
```

Constants become available to use in scripts after compilation and updating of assembly of metadata.

System constants

The system constants are described in *UltimaConstants* class:
- *CompanyName*;
- *Copyright*;
- *Trademark*;
- *FullVersionString*.

IDictionaryCommandManager

The manager (from namespace *Ultima.Scripting*) is intended for work with commands on dictionary record.

The interface of *IDictionaryCommandManager* implements the following methods:
- *GetDictionaryCommands(long dictionaryId)* – returns the list of commands which can be carried out over record of the specified dictionary:
  - *dictionaryId* – dictionary identifier;

- *GetCommandsScriptHotkeys(long dictionaryId)* – returns the list of hot keys for calling commands, which can be carried out over record of the specified dictionary:
  - *dictionaryId* – dictionary identifier;
- *ExecuteCommand(long id, long recordId, IDictionary<string, object> parameters)* – executes the command on the specified dictionary record, returns the list ClientActions to perform on the client side:
  - *id* – identifier of the dictionary record command;
  - *recordId* – dictionary record ID;
  - *parameters* – parameters of command execution.

IDictionaryListCommandManager

The manager (from namespace *Ultima.Scripting*) is intended for work with dictionary list commands (list of records).

The interface of *IDictionaryListCommandManager* implements the following methods:

- *GetDictionaryListCommands(long dictionaryId)* – returns the list of commands which can be carried out over list of records record of the specified dictionary:
  - *dictionaryId* – dictionary identifier;
- *GetCommandsScriptHotkeys(long dictionaryId)* – returns the list of hot keys for calling commands, which can be carried out over the list of records of the specified dictionary:
  - *dictionaryId* – dictionary identifier;
- *GetDictionaryCaption(long scriptId)* – returns the description of the dictionary for a script of command which is executed over the list of its records:
  - *scriptId* – script command identifier;
- *ExecuteCommand(long id, long records, IDictionary<string, object> parameters)* – executes the command on the specified dictionary records, returns the list ClientActions to perform on the client side:
  - *id* – identifier of the dictionary record command;
  - *records* – dictionary records identifiers;
  - *parameters* – parameters of command execution.

IDocumentCommandManager

The manager (from namespace *Ultima.Scripting*) is intended for work with commands on the document.

The interface of *IDocumentCommandManager* implements the following methods:

- *GetDocumentCommands(long docSubtypeId)* – returns the list of commands which can be carried out over the document of the specified subtype:
  - *docSubtypeId* – document subtype identifier;
- *GetCommandsScriptHotkeys(long docSubtypeId)* – returns the list of hot keys for calling commands, which can be carried out over the document of the specified subtype:
  - *docSubtypeId* – document subtype identifier;
- *ExecuteCommand(long id, long documentId, IDictionary<string, object> parameters)* – executes the command on the specified document, returns the list ClientActions to perform on the client side:
  - *id* – identifier of the document command;
  - *documentId* – document ID;
  - *parameters* – parameters of command execution.

IDocumentListCommandManager

The manager (from namespace *Ultima.Scripting*) is intended for work with commands on the list of documents.

The interface of *IDocumentListCommandManager* implements the following methods:

- *GetDocumentListCommands(long docTypeId)* – returns the list of commands which can be carried out over the list of documents of the specified subtype:
  - *docTypeId* – document type ID;
- *GetCommandsScriptHotkeys(long docTypeId)* – returns the list of hot keys for calling commands, which can be carried out over the list of documents of the specified type:
  - *docTypeId* – document type ID;
- *ExecuteCommand(long id, long[] documents, IDictionary<string, object> parameters)* – executes the command on the specified documents, returns the list ClientActions to perform on the client side:
  - *id* – identifier of the documents list command;
  - *documents* – document IDs;
  - *parameters* – parameters of command execution.

IEmailService

The manager (from namespace *Ultima*) is intended for work with service of sending e-mail messages.

Manager interface *IEmailService* realizes methods, sending messages (e-mail):

- *SendMail(EmailMessage message, EmailOptions emailOptions = null)* — an asynchronous method for sending messages*:*
  - *message* — message for sending. It contains the following properties and methods:
    - *AddressFrom* — sender's address;
    - *AddressFromName* — sender' name;
    - *AddressTo* — destination address;
    - *AddressToName* — recipient name;
    - *Subject* — heading;
    - *Body* — letter text;
    - *IsHtmlBody* — flag, indicating that the message body is transmitted in Html format;
    - *AttachFile* — adds the file attachment to the message;
    - *EmbedFile* — builds in the specified file of attachment (for example, the image) into the message body.
  - *emailOptions* — settings of sending of the message (optional parameter), contains the following properties:
    - *SmtpServerID* — server code from the dictionary of SMTP servers;
    - *SendOnRollback* — sends the message, only if the current transaction is cancelled;
    - *ScheduledSendingTime* — time of message sending;
    - *AllowedSendIntervalFrom* — lower bound of an allowable sending interval (time of a day);
    - *AllowedSendIntervalTo* — upper bound of an allowable sending interval (time of a day).

Outdated method:

- *SendMail(string addressTo, string addressToName, string subject, string body, string addressFrom, string addressFromName, bool isHtmlBody = false, string[] attachementFiles = null, string[] embeddedFiles = null)* — a synchronous method of sending:
  - *addressTo* – e-mail of the letter recipient;
  - *AddressToName* — letter recipient name;
  - *subject* – letter subject;
  - *body* — letter body;
  - *addressFrom* – e-mail of the letter sender;
  - *AddressFromName* — letter sender's name;
  - *isHtmlBody* –the flag, specifying that the text in the body of the letter has HTML format;
  - *attachementFiles* – the attached files (names of files at a disk);
  - *embeddedFiles* – built in (in a letter body) files (names of files at a disk).

  This method sends the message synchronously, blocking the current transaction for the period of sending.

  It is marked as outdated and forbidden to use.

IExportManager

The manager (from *Ultima* namespace) is designed to work with print form export service.

The interface of *IExportManager* implements the following methods:

- *Export(ExportFormat options, long printFormId, PrintFormParameters parameters)* exports the specified print form, not associated with any object, into the specified format:
  - *options* – a format, which the export is carried out into, can assume values:
    - *Pdf* – pdf format;
    - *Xls* – excel format;
    - *Xlsx* – excel format;
    - *Rtf* – rtf format;
    - *Html* – html format;
    - *Image* – image;
  - *printFormId* – print form ID;
  - *parameters* – additional print parameters ;
- *ExportDictionaryRecord(Type dictionaryType, long recordId, long printFormId, ExportFormat options, IDictionary<string, object> parameters = null)* exports the print form of specified dictionary record into the specified format:
  - *dictionaryType* – dictionary type;
  - *recordId* – dictionary record ID;
  - *printFormId* – print form ID;
  - *options* – a format, which the export is carried out into;
  - *parameters* – additional print parameters (default value is null);

  There is also typed version of this method *ExportDictionaryRecord<T>(this IExportManager manager, long recordId, long printFormId, ExportFormat options, IDictionary<string, object> parameters = null)*;
- *ExportDictionaryList(Type dictionaryType, long[] recordList, long printFormId, ExportFormat options, IDictionary<string, object> parameters = null)* exports the print form of specified dictionary records into the specified format:
  - *dictionaryType* – dictionary type;
  - *recordId* – a list of IDs of dictionary records;
  - *printFormId* – print form ID;
  - *options* – a format, which the export is carried out into;
  - *parameters* – additional print parameters (default value is null);

  There is also typed version of this method *ExportDictionaryList<T>(this IExportManager manager, long[] recordList, long printFormId, ExportFormat options, IDictionary<string, object> parameters = null)*;
- *ExportDocument(long documentId, long printFormId, ExportFormat options, IDictionary<string, object> parameters = null)* exports the print form of specified document into the specified format:
  - *documentId* – document ID;
  - *printFormId* – print form ID;
  - *options* – a format, which the export is carried out into;
  - *parameters* – additional print parameters (default value is null);
- *ExportDocumentList(long[] documentList, long printFormId, ExportFormat options, IDictionary<string, object> parameters = null)* exports the print form of specified documents into the specified format:
  - *documentList* – a list of documents IDs;
  - *printFormId* – print form ID;
  - *options* – a format, which the export is carried out into;
  - *parameters* – additional print parameters (default value is null);

Example of use of API export:

```
// values of parameters
long recordId = 1;
long documentId = 1;
long printFormId = 1;
var parameters = new Dictionary<string, object>();
parameters["Param1"] = "test";
parameters["Param2"] = "parameter";

// export the print form of dictionary record into pdf format
var action1 = new SaveFileAction("exp.pdf",
       ExportManager.ExportDictionaryRecord<Language>(recordId,
       printFormId, ExportFormat.Pdf, parameters));

// export the print form of the document into rtf format
var action2 = new SaveFileAction("exp.rtf",
       ExportManager.ExportDocument(documentId,
       printFormId, ExportFormat.Rtf, parameters));
```

ILinkTableManager

The manager (from namespace *Ultima.Dictionaries*) is designed to handle link tables.

The interface of *ILinkTableManager* implements the following methods:

- *GetRecords(Type linkTableType, LambdaExpression selectExpression, LambdaExpression filterExpression)* returns a table of link table records meeting the filter condition:
  - *linkTableType* – link table type;
  - *selectExpression* – an expression describing which of link table columns values will be loaded. If null is indicated as parameter value – the values of all columns will be loaded;
  - *filterExpression* – an expression describing which dictionary record will be loaded (see details in the section Filters);
- *GetRecords(Type linkTableType, LambdaExpression selectExpression, string dictionaryKeyName, long dictionaryKeyValue)* returns a table of link table records meeting the specified values:
  - *linkTableType* – link table type;
  - *selectExpression* – an expression describing which of link table columns values will be loaded. If null is indicated as parameter value – the values of all columns will be loaded;
  - *dictionaryKeyName* – a name of dictionary property for filtration;
  - *dictionaryKeyValue* – a value of dictionary property for filtration;
- *SaveRecords(Type linkTableType, ILinkTable recordCollection)* saves the collection of link table records:
  - *linkTableType* – link table type;
  - *recordCollection* – a table of records being saved;
- *SaveRecord(Type linkTableType, ILinkTableRecord record)* saves the link table record:
  - *linkTableType* – link table type;
  - *record* – a record being saved;
- *DeleteRecords(Type linkTableType, ILinkTable recordCollection)* deletes the collection of link table records:
  - *linkTableType* – link table type;
  - *recordCollection* – a table of records being deleted;
- *DeleteRecords(Type linkTableType, LambdaExpression filterExpression)* deletes the link table records meeting the filter condition:
  - *linkTableType* – link table type;
  - *filterExpression* – an expression describing which dictionary record is deleted (see details in the section Filters).

There is also typed *ILinkTableManagerT* class with similar methods:

```
LinkTableManager.SaveRecords(typeof(price list), list);

LinkTableManager<price list>.SaveRecords(list);
```

ILogger and ILogManager

Manager ILogger(from namespaceUltima.Log) is intended for use functional libraries Serilog, through which central logging is implemented.

Manager interface*ILogger* consist the set of methods for logging events with different levels of importance: *Debug()*, *Info()*, *Verbose()*, *Warn()*, *Error()* и *Fatal()*.

For import of service it is recommended to use ILogger <T>interface parametrized by class user type. If ILogger without parameter is imported, the log entry is added on behalf of «Server» and if ILogger <MyClass> is imported — that record will be on behalf of «MyClass» Such records easier to search, filter and group by class source:

```
[Import]
private ILogger<MyService> Logger { get; set; }
```

Serilog uses simple DSL to define names for additional properties of logging event. Synthaxis of DSL is similar to formating lines for method *string.Format*:

```
Logger.Debug("Saving document {DocumentID}, created by {UserID}", docId, userId);
```

As a result of this call the event with two properties of scalar type will log: *DocumentID* and *UserID*. Syntaxstring.Format with numbered parameters like {DocumentID} is also supported.

Named logger can be received by usual way:

```
LogManager.GetLogger(”MyLoggerName”);
```

without *ILogManager* (from namespace*Ultima.Log*):

```
var newLogger = Logger.Named("NewLoggerName");
```

The more interested example of usinf Serilog - is saving the properties of structural type (provides arrays, classes, structures, hash-table, dictionaries and documents):

```
var parameters = new Dictionary<string, object>();
parameters["id"] = id;
parameters["limit"] = maxValues;
...

Logger.Verbose("Executing command: {SQL:l} with parameters: {@Parameters}",
       sqlCommand, parameters);
```

In this case all parameters list will be added to the logging event and record about of command with definite value of parameter can be found in log.

While logging exclusions their inner structure will saved, including HResult, CallStack properties and InnerException chain. To log the exclusion,you need to transfer it as the first parameter to logging methods:

```
Logger.Error(ex, "Cannot execute user task: {Message}", ex.Message);
```

Exclusion loging methods let adding any additional parameters to the message text.

Special destructurizator added for dictionary and documents logging. It serves as standard, but cuts over information from objects: Descript of class, property IsPropertyChanged and i.e.:

```
Logger.Debug(”Brand #{BrandID} contents: {@Brand}", brand.ID, brand);

// message in log
// Brand #12 contents: Brand { ID: 12, Name: "Pineapple" }
```

You should carefully use and don’t abuse of document logging as documents usually is heavier than dictionaries, and saving them into the journal cause ending the resources of storage. Typical result of document logging is:

```
Document #101187 is saved. Document contents: SaleDocument { ActualInvoiceDocumentID:
null, AgentID: 636, AllowPartialRelease: False, Amount: 5500, AmountDistributionTypeID:
null, ArticlesQuantity: 3, ClientDueOnDelivery: 5500, Comments: “noreply”, Convertation:
null, CreationDate: 02/06/2015 22:21:53, CreatorID: 7, CustomerSupplyContractID: null,
DeadDate: 02/11/2015 22:21:53, Deleted: False, DeliveryActive: False, Description: "Sales
(Picking) #101187, 2/6/2015", ExplicitDeadDate: False, FirmID: 14, ID: 101187,
InterstoreResupply: False, KickbackAgentID: null, KickbackAmount: null, KickbackTypeID:
null, ManagerID: null, OfficeID: 38, PickupDate: null, PowerOfAttorneyID null,
PriceTypeID: 1, PriceZoneID: 6, RowsCount: 5, SaleTerminalID: null, StoreID: 522,
SubtypeID: 4131, TotalsList: "4251", TransactionDate: 02/06/2015 22:21:53, TypeID: 4111,
Version: 12, Volume: 0.001442, Weight: 2.70, ActualInvoiceDocument: null, Agent: null,
AmountDistributionType: null, Creator: null, CustomerSupplyContract: null, Firm: null,
KickbackAgent: null, KickbackType: null, Manager: null, Office: null, PowerOfAttorney:
null, PriceType: null, PriceZone: null, SaleTerminal: null, Store: null, Subtype: null,
Type: null, ArticleBarcodes: [], ArticleCcds: [ArticleCcdTablePartRow { ArticleID: 491967,
CcdID: 1, Checked: False, Deleted: False, DocumentDeleted: False, DocumentID: 101187, ID:
5686087, Quantity: 1, TablePartEntryID: 12422, TransactionDate: 02/06/2015 22:21:53,
Article: null, Ccd: null, Document: null, TablePartEntry: null }, ArticleCcdTablePartRow
{ ArticleID: 491964, CcdID: 53, Checked: False, Deleted: False, DocumentDeleted: False,
DocumentID: 101187, ID: 5686088, Quantity: 2, TablePartEntryID: 12422, TransactionDate:
02/06/2015 22:21:53, Article: null, Ccd: null, Document: null, TablePartEntry: null }],
Articles: [SaleArticleTablePartRow { Amount: 3300, ArticleID: 491967, Checked: False,
Deleted: False, DocumentDeleted: False, DocumentID: 101187, ID: 5686089,
NotReservedQuantity: 0, OriginalPrice: 3300, ReservedQuantity: 1, ResupplyQuantity: 0,
SalePrice: 3300, SaleQuantity: 1, TablePartEntryID: 5229, TransactionDate: 02/06/2015
22:21:53, Article: null, Document: null, TablePartEntry: null }, SaleArticleTablePartRow
{ Amount: 2200, ArticleID: 491964, Checked: False, Deleted: False, DocumentDeleted: False,
DocumentID: 101187, ID: 5686090, NotReservedQuantity: 0, OriginalPrice: 1100,
ReservedQuantity: 2, ResupplyQuantity: 0, SalePrice: 1100, SaleQuantity: 2,
TablePartEntryID: 5229, TransactionDate: 02/06/2015 22:21:53, Article: null, Document:
null, TablePartEntry: null }, SaleArticleTablePartRow { Amount: 0, ArticleID: 492060,
Checked: False, Deleted: False, DocumentDeleted: False, DocumentID: 101187, ID: 5686091,
NotReservedQuantity: 1, OriginalPrice: 8700, ReservedQuantity: 0, ResupplyQuantity: 0,
SalePrice: 8700, SaleQuantity: 0, TablePartEntryID: 5229, TransactionDate: 02/06/2015
22:21:53, Article: null, Document: null, TablePartEntry: null }, SaleArticleTablePartRow
{ Amount: 0, ArticleID: 492075, Checked: False, Deleted: False, DocumentDeleted: False,
DocumentID: 101187, ID: 5686092, NotReservedQuantity: 1, OriginalPrice: 8700,
ReservedQuantity: 0, ResupplyQuantity: 0, SalePrice: 8700, SaleQuantity: 0,
TablePartEntryID: 5229, TransactionDate: 02/06/2015 22:21:53, Article: null, Document:
null, TablePartEntry: null }, SaleArticleTablePartRow { Amount: 0, ArticleID: 492077,
Checked: False, Deleted: False, DocumentDeleted: False, DocumentID: 101187, ID: 5686093,
NotReservedQuantity: 1, OriginalPrice: 8600, ReservedQuantity: 0, ResupplyQuantity: 0,
SalePrice: 8600, SaleQuantity: 0, TablePartEntryID: 5229, TransactionDate: 02/06/2015
22:21:53, Article: null, Document: null, TablePartEntry: null }], Delivery: [],
DeliveryWizard: [], DistribKickback: [], ExcludedArticles: [] }
```

If the document needs to be saved in log, but you don't want to duplicate the content in text message, you can add it only to the properties and it will not mention in the text of message. You can do it using the method *With*:

```
var doc = DocumentManager.GetDocument(documentId);
Logger.With("Document", doc).
       Debug(@"Document #{DocumentID} is saved. Document contents: {$DocTitle}",
       doc.ID, doc);

// message in log:
// Document #101187 is saved. Document contents: "Sales (Picking) #101187, 2/6/2015"
```

Using this method you can mark log events group, relating to one subject. For examples, in recounting of prices you can add the good code to each message:

```
logger = Logger.With("ArticleID", 123);
logger.Info("Recalculating prices...");
logger.Warn(ex, "Can't recalculate prices due to {Cause}", ex.Message);
```

The last way - adding the property to all messages in block *using*. It let to transfer logger back and forth between methods and services:

```
//to all events in block using ArticleID property will be added
using (LogManager.With("ArticleID", article.ID))
{
       Logger.Debug("Processing article: {ArticleName}", article.Name);
       PriceService.RecalculatePrices(article.ID);
       DescriptionService.ValidateDescription(article.ID);
       PhotoService.MakePhotos(article.ID);
       Logger.Debug("Done with article: {ArticleName}", article.Name);
}
```

INotificationService

The manager for working with *INotificationService* realizes the following methods:

- *NotifyUserOnSuccess(long toUserId, long notifyCategoryId, string title, string text, ClientAction action, NotificationIcon icon)* — sends a notification after the successful completion of the server call:
  - *toUserId* — user ID to which the message is sent;
  - *notifyCategoryId* — category notification identifier;
  - *title* — notification title;
  - *text* — notification text;
  - *action* — ClientAction, transferred with the notification which can be executed by the user;
  - *icon* — notification icon (by default the notification is sent without icons).
- *NotifyUserOnFail(long toUserId, long notifyCategoryId, string title, string text, ClientAction action, NotificationIcon icon)* — sends a notification after interruption of a server call by any exception:
  - *toUserId* — user ID to which the message is sent;
  - *notifyCategoryId* — category notification identifier;
  - *title* — notification title;
  - *text* — notification text;
  - *action* — ClientAction, transferred with the notification which can be executed by the user;
  - *icon* — notification icon (by default the notification is sent without icons).
- *NotifyUserImmediately(long toUserId, long notifyCategoryId, string title, string text, ClientAction actionNotificationIcon icon)* — sends a notification immediately:
  - *toUserId* — user ID to which the message is sent;
  - *notifyCategoryId* — category notification identifier;
  - *title* — notification title;
  - *text* — notification text;

- *action* — ClientAction, transferred with the notification which can be executed by the user;
- *icon* — notification icon (by default the notification is sent without icons).

- *MarkNotificationsRead(IDList notificationList)* — marks the specified notifications as read:
  - *notificationList* — ids notification list ;
- *GetUnreadNotificationsCount(long? userId = null)* — returns the number of notifications unread by the user:
  - *userId* — user id. If to specify null as value of parameter — the number of unread notifications of the current user will be returned;
- *GetUnreadNotifications()* — returns the list of identifiers of unread notifications;
- *GetNotifications(DateTime? fromDate, DateTime? toDate, long? categoryId = null, bool unreadOnly = true, long? userId = null)* — returns the list of notifications that meet parameters:
  - *fromDate* — start of the time interval in which the notification was sent;
  - *toDate* — end of the time interval in which the notification was sent;
  - *categoryId* — category notification identifier. If to specify null as value of parameter — the notifications of all categories will be returned;
  - *unreadOnly* — if to specify *true*, as the value of parameter, only unread notifications will be returned;
  - *userId* — user ID, to which notifications were sent. If to specify null as value of parameter — the notifications sent to the current user will be returned;
- *GetCategoryName(long categoryId)* — returns the name of the specified category of notifications:
  - *categoryId* — category notifications identifier.
- *EventHandler<NotificationEventArgs> NotificationSent* — event handler, executed after the notification sending;
- *EventHandler<NotificationEventArgs> NotificationRead* — event handler, executed after the notification reading.

IPrintManager

A batch print queue is implemented in the system Ultimate AEGIS®. Namely, several print tasks as a single package may be sent to print. This means that the tasks sent will be printed in the given order, and no foreign tasks will be squeezed between them. Thus, to print from the handler one will need to create a package and add needed tasks into it. The tasks themselves can be either delayed (the print form data generation handler will be called in asynchronous manner from another transaction) or synchronous – the handler will be called immediately. The latter option makes the execution of a server call longer, however, if there is a risk of change in data before the task has been printed, the synchronous option would be better to use.

The interface of manager *IPrintManager* (from the *Ultima.Printing* namespace) impements the following methods:

- *CreatePackage(long printerId)* creates a new print package *PrintPackage* for the printer specified:
  - *printerId* – printer's ID;

  On the client side, this package is to be completed with tasks by methods of the class *PrintPackage*:

```
var pack = PrintManager.CreatePackage(printerId);
pack.AddDictionaryRecordTask(typeof(Good), goodId, printFormId, copies, parameters);
PrintManager.Print(pack);
```

- *Print(PrintPackage pack)* – synchronous print of the completed package – the*GetData* method for all tasks is called, and the package gets added to the print queue:
  - *pack* – print package;
- *PrintDelayed(PrintPackage pack)* – asynchronous print – the package gets added to the print queue:
  - *pack* – print package;

The interface implements methods for single tasks as well. These are used when it is not necessary to create a package (in fact, a package including a single task will be created anyway). Each method is implemented in two versions: for synchronous and asynchronous prints. While in synchronous printing, an object with print form data is immediately calculated and created (the client awaits the completion of the process). While in asynchronous (delayed) printing, the data will be created afterwards:

- *Print(long printerId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – a synchronous print of a print form not related to a particular object:
  - *printerId* – printer's ID;
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);
- *PrintDelayed(long printerId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – a method of asynchronous print similar to the previous one;
- *PrintDictionaryRecord(long printerId, Type dictionaryType, long recordId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – synchronous print of a (single) dictionary record:
  - *printerId* – printer's ID;
  - *dictionaryType* – dictionary type;
  - *recordId* – dictionary record ID;
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);

  There is also a typified version of this method: *PrintDictionaryRecord<T>(this IPrintManager manager, long printerId, long recordId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)*;
- *PrintDictionaryRecordDelayed(long printerId, Type dictionaryType, long recordId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – a method of asynchronous print similar to the previous one.

  There is also a typified version of this method: *PrintDictionaryRecordDelayed<T>(this IPrintManager manager, long printerId, long recordId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)*;
- *PrintDictionaryList(long printerId, Type dictionaryType, IDList recordList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – synchronous print of a dictionary records list:
  - *printerId* – printer's ID;
  - *dictionaryType* – dictionary type;
  - *recordList*– a list of dictionary records IDs;
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);

  There is also a typified version of this method: *PrintDictionaryList<T>(this IPrintManager manager, long printerId, IDList recordList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)*;
- *PrintDictionaryListDelayed(long printerId, Type dictionaryType, IDList recordList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – a method of asynchronous print similar to the previous one.

  There is also a typified version of this method: *PrintDictionaryListDelayed<T>(this IPrintManager manager, long printerId, IDList recordList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)*;
- *PrintDocument(long printerId, long documentId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – synchronous print of a (single) document:
  - *printerId* – printer's ID;
  - *documentId* – document ID;
  - *printFormId* – print form ID;

- *copies* – number of printed copies (default value is 1);
- *parameters* – additional print parameters (default value is null);

- *PrintDocumentDelayed(long printerId, long documentId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – a method of asynchronous print similar to the previous one;
- *PrintDocumentList(long printerId, IDList documentList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – synchronous print of a documents list:
  - *printerId* – printer's ID;
  - *documentList* – a list of documents IDs;
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);
- *PrintDocumentListDelayed(long printerId, IDList documentList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – a method of asynchronous print similar to the previous one.

Regardless of how a print task is sent to the printer – either synchronous or delayed – the real sending to the print server will occur only after the server call, from which the task was sent, has been successfully completed. If the server call is completed with an error, no sending will occur. To secure the sending of a task, one needs to create a separate transaction, send the task from it and then to commit the transaction.

Other methods implemented by the interface are intended to manage the print queue and execute service tasks:

- *GetPrintersList(long printServerId)* – returns the list of printers' system names for the print server specified:
  - *printServerId* – print server's ID;
- *SavePrintStatistics(long userId, string printerName, long printFormId, long versionId, long pages, long copies)* – saves print statistics in the database:
  - *userId* – ID of the user, who printed the task;
  - *printerName* – name of printer, where the task was printed;
  - *printFormId* – ID of print form used;
  - *versionId* – metadata version ID;
  - *pages* – number of pages printed;
  - *copies* – number of copies printed;
- *BuildNextTask()* – fills in the print form of the next queued print task with the data and renders it, if no data available, i. e. if the task was created by means of the asynchronous method;
- *SendNextTaskToPrint()* – sends the next queued print task to print;
- *MoveTaskToErrorState(long taskId, string errorData)* – moves the task into print fault status:
  - *taskId* – print task ID;
  - *errorData* – error data;
- *PrinterAlarm(long printerId)* – informs the admin on a printer problem:
  - *printerId* – printer's ID;
- *PrinterAlarm(string systemName)* – informs the admin on a printer problem:
  - *systemName* – printer's system name;
- *PrintServerAlarm(long printServerId)* – informs the admin on a print server problem:
  - *printServerId* – print server's ID;
- *GetPrintServerID(long printerID)* – returns the print server ID for the printer specified:
  - *printerID* – printer's ID;
- *GetFilteredPackageQueue(IDList packageList, PrintPackageFilter filter, int rowsCountLimiter)* – returns the print packages queue, which satisfies the conditions of the filter:
  - *packageList* – list of packages IDs. If to put null as a parameter's value, a filter search will be performed among all print packages;

- *filter* – filter *PrintPackageFilter* that describes which packages are to be returned. Class *PrintPackageFilter* – data container for the values of the filter applied to print packages – possesses the following properties:
  - *FailedOnly*, of *bool* type – if to specify true as a parameter's value, a filter search will be performed only among print packages with the failed status;
  - *StateID*, of *long* type – print package status ID;
  - *SessionID*, of *string* type – GUID of session;
  - *PrintTimeFrom*, of *DateTime* type – date of the beginning of the period, when the print package was created;
  - *DateTime*, of *DateTime* type – date of the ending of the period, when the print package was created;
  - *PrintServerID*, of *long* type – print server ID;
  - *PrinterID*, of *long* type – printer ID;
  - *UserID*, of *long* type – user ID;

  If to specify null as a parameter's value, the search will be performed with the empty filter;
- *rowsCountLimiter* – returned rows limiter;

- *SetPackageState(long packageId, long packageStateId)* – sets a status for a print package specified:
  - *packageId* – print package ID;
  - *packageStateId* – package status ID;
- *SetPrinterPackagesState(long printerId, long packageStateId)* – sets a status for all print packages for the printer specified:
  - *printerId* – printer's ID;
  - *packageStateId* – print package status ID;
- *SetPrintServerPackagesState(long printServerId, long packageStateId)* – sets a status for all print packages for the print server specified:
  - *printServerId* – print server's ID;
  - *packageStateId* – print package status ID;
- *SetPackagePrinter(long packageId, long printerId)* – sets a printer for the print package specified:
  - *packageId* – print package ID;
  - *printerId* – printer's ID;
- *MovePrinterPackagesToPrinter(long printerId, long newPrinterId)* – moves all print packages from one printer specified to another:
  - *printerId* – ID of the printer, from which the packages are to be transferred;
  - *newPrinterId* – ID of the printer, to which the packages are to be transferred;
- *CleanupPackage(long packageId)* – remove the print package specified:
  - *packageId* – print package ID;
- *CleanupPrinterPackages(long printerId)* – remove all print packages for the printer specified:
  - *printerId* – printer's ID;
- *CleanupPrintServerPackages(long printServerId)* – remove all print packages for the print server specified:
  - *printServerId* – print server's ID;
- *EventHandler PrintTasksAvailable* – event handler implemented after available print tasks has been created.

Class *PrintPackage* (from *Ultima.Printing* namespace) – data container used to fill in the queued packages with print tasks – implements the following methods:

- *AddTask(long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* adds a print task for the print form, not associated with any object:
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);

- *AddDictionaryRecordTask(Type dictionaryType, long recordId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – adds a print task of a (single) dictionary record:
  - *dictionaryType* – dictionary type;
  - *recordId* – dictionary record ID;
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);

  There is also a typified version of this method:*AddDictionaryRecordTask<T>(this PrintPackage pack, long recordId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)*;
- *AddDictionaryListTask(Type dictionaryType, IDList recordList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – adds a dictionary list print task:
  - *dictionaryType* – dictionary type;
  - *recordList*– a list of dictionary records IDs;
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);

  There is also a typified version of this method: *AddDictionaryListTask<T>(this PrintPackage pack, IDList recordList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)*;
- *AddDocumentTask(long documentId, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – adds a print task for a (single) document:
  - *documentId* – document ID;
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);
- *AddDocumentListTask(IDList documentList, long printFormId, long copies = 1, IDictionary<string, object> parameters = null)* – adds a document list print task:
  - *documentList* – a list of documents IDs;
  - *printFormId* – print form ID;
  - *copies* – number of printed copies (default value is 1);
  - *parameters* – additional print parameters (default value is null);

Example of API print use:

```
// values of parameters
long printerId = 1;
long recordId = 1;
long printFormId = 1;
long copies = 2;
long documentId = 1;
var parameters = new Dictionary<string, object>();
parameters["Param1"] = "Hello";
parameters["Param2"] = "world!";

// print package creation
var pack = PrintManager.CreatePackage(printerId);

// complete package with tasks
pack.AddDictionaryRecordTask(typeof(Language), recordId, printFormId, copies, parameters);
pack.AddDictionaryRecordTask<Language>(recordId, printFormId, copies, parameters);
pack.AddTask(printFormId, copies, parameters);

// send package to printer
PrintManager.Print(pack);

// implicit use of package convenient for sending a single print task
PrintManager.PrintDocument(printerId, documentId, printFormId, copies, parameters);
```

```
// default values for copies = 1 and parameters = null
PrintManager.PrintDocument(printerId, documentId, printFormId);
```

ISmsService

The manager (from namespace *Ultima*) is intended for work with service of sending of short messages (SMS).

The interface of *ISmsService* implements the following methods:

- *SendMessage*(string phone, string message, SmsOptions smsOptions = null) — sends the message asynchronously:
  - *phone* — mobile phone number;
  - *message* — message text.
  - *smsOptions* — settings of sending of the message, optional parameter.. It contains the following properties:
    - *SmppServerID* — server code from the dictionary of SMPP servers;
    - *SendOnRollback* — sends the message, only if the current transaction is cancelled;
    - *ScheduledSendingTime* — time of message sending;
    - *AllowedSendIntervalFrom* — lower bound of an allowable sending interval (time of a day);
    - *AllowedSendIntervalTo* — upper bound of an allowable sending interval (time of a day).

Outdated method:

- *Send(string phone, string message)* – synchronously sends the message to the specified telephone number:
  - *phone* — mobile phone number;
  - *message* – message text.

  This method blocks the current transaction for the period of sending of the message.
  It is marked as outdated and forbidden to use.

ITotalsManager

The manager (from namespace *Ultima.Totals*) is designed to handle totals.

The interface of *ITotalsManager* implements the following methods:

- *SaveDocumentTransactions(IDList documentList, Dictionary<long, TotalLimitDecreaseRequest> oldTotalLimits, TransactionPairCollection transactionPairs, TransactionCollection transactions)* – keeps transactions of the specified documents:
  - *documentList* – a list of documents IDs;
  - *oldTotalLimits* – old limit of results for all documents;
  - *transactionPairs* – collection of the pair of transactions for balance totals;
  - *transactions* – collection of the transactions for non-balance totals;
- *DeleteDocumentTransactions(IDList documentList)* – deletes transactions of the specified documents:
  - *documentList* – a list of documents IDs;
- *BuildReport(ReportSetup setup)* – build the report:
  - *setup* – report parameters. The format of parameters is defined by a class-container *ReportSetup*;
- *CalculateTotals()* – recalculate totals;
- *PostTotalLimitDecreaseQueue(DateTime limitDate, long? limitDocumentId = null)* – request for reduction of a limit of results before the specified date or, optionally, the document:
  - *limitDate* – the date until which the limit of totals should be reset;
  - *limitDocumentId* – document, before the date of which it is necessary to reset limit totals (optional parameter);

IUserCommandManager

The manager (from namespace *Ultima.Scripting*) is intended for work with the user commands.

The interface of *IUserCommandManager* implements the following methods:

- *GetUserCommands()* – returns the list of all user commands;
- *GetAvailableUserCommands()* – returns the list of command identifiers, available to the current user;
- *ExecuteUserCommand(long id, IDictionary<string, object> parameters)* – executes the specified user command, returns the list ClientActions to perform on the client side:
  - *id* – user command ID;
  - *parameters* – parameters of command execution.

IUserManager

The manager (from the namespace *Ultima*) is designed to handle users.

The interface of *IUserManager* implements the following methods and has the following properties:

- *CurrentUserID*, type *long* returns current user ID;
- *CurrentUser*, type *UltimaIdentity* returns current user;
- *CurrentCulture*, type *CultureInfo* returns culture of current user;
- *GetUserCulture(long userId)*, type *CultureInfo* returns *CultureInfo* culture of specified user:
  - *userId* – user ID;
- *GetUsers()*, type *IList<User>* returns a list of all users in the format: *ID*, *name* and *login*;
- *RenewPassword(string oldPasswordHash, string newPasswordHash)* changes the password for current user provided for that the hash sum of existing password coincides with the password hash sum stored in the database:
  - *oldPasswordHash* – hash sum of existing password;
  - *newPasswordHash*– hash sum of new password;
- *GetSystemUsers()*, type *IDList* returns a list of IDs of all system users. In the dictionary Users, these users are located in the group *System* and are used for execution of official queries to the database without direct participation of company employees, e.g. to work with the print server and server for export, initiation of tasks, etc.

Examples of use:

```
// get current user ID
var userId = UserManager.CurrentUserID;
```

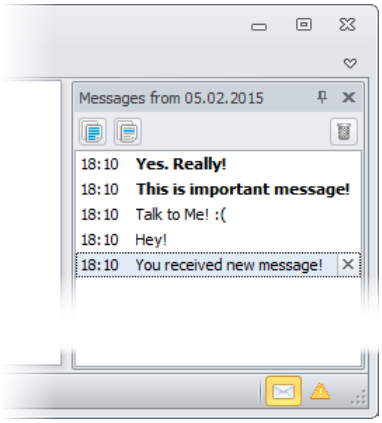
IUserMessages

The manager (from namespace *Ultima.Client*) is designed to handle user messages.

The interface of *IUserMessages* implements the following method:

- *CreateUserMessage(UserMessageImportance importance, string format, params object[] parameters)* creates a message for current user in the current session:
  - *importance* – importance of the message (optional). Enumeration *UserMessageImportance* has the following values:
    - *Normal* – normal message;
    - *Important* – important message (**bolded** in the list).

    If parameter *importance* is not used, the message will be created with *Normal* importance;
  - *format* – a text of the message with (optional) elements of the type format *{0}*, *{1}* etc., which will be replaced with text equivalents of the values of corresponding objects;
  - *parameters* – an array of objects (optional) to replace the existing elements of the format.



Example of use:

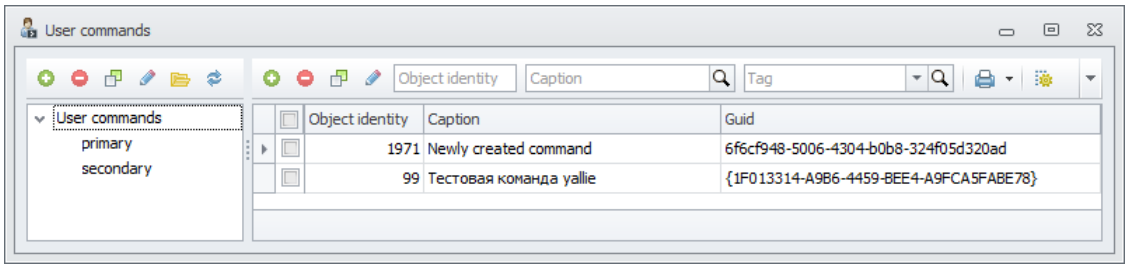
```
// values of parameters
object[] msgParams = { "customer", 100 };

// create a message of normal importance
UserMessages.CreateUserMessage("User {0} has {1} bonuses", msgParams);

// create a message of high importance
UserMessages.CreateUserMessage(UserMessageImportance.Important, "All your bonuses are
reset to zero!");
```

Interactive commands

Interactive commands are requested immediately by a user.

User commands

Custom commands — these are the scripts that implement an interface *IUserCommand* that can be added to the main menu of the client application, respectively, running thence by the user . The list of all user settings can be found in the form "User commands":

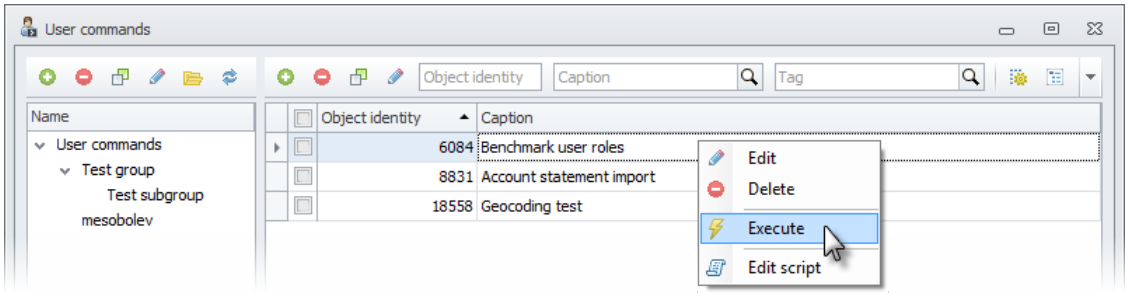


Dictionary window divided into two parts: a tree of the group of commands is displayed to the left, a list of commands selected on the left of the group is displayed to the right.

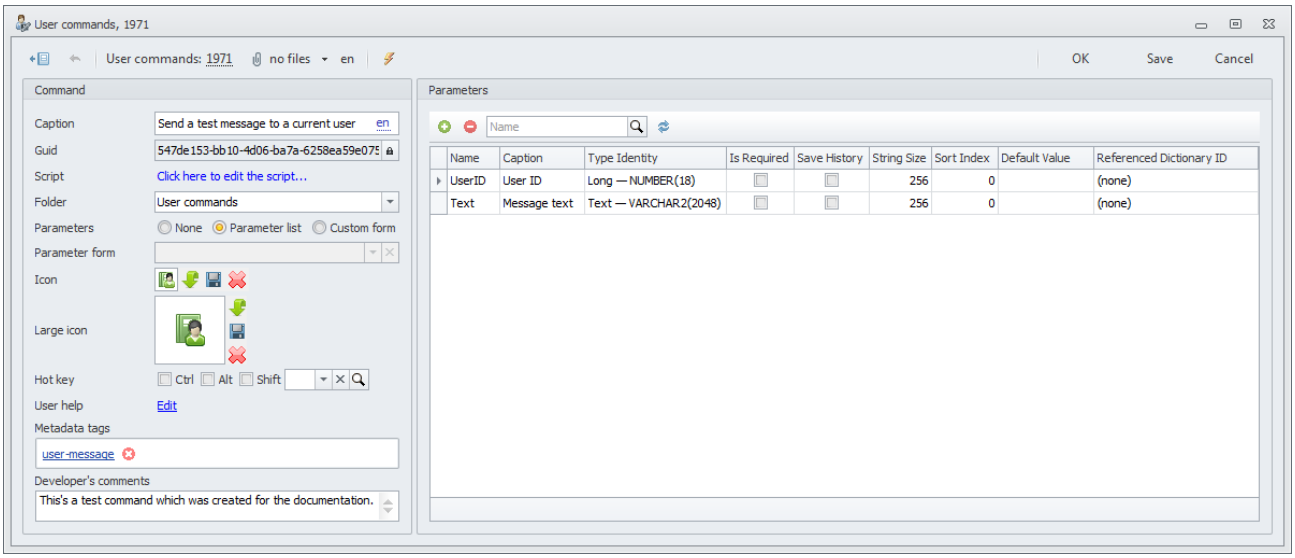
Dictionary records can be filtered by *shown on screen forms names* commands (*Caption*) and Tags(*Tag*).

The script of the user command selected in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

Selected command can be executed by selecting *Execute* in this context menu:

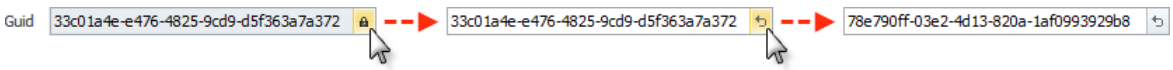


Button on the tool panel of the form of editing the user command execute the same function and let to execute the command immediately:



The user command has the following properties:

- *Caption* – command name displayed in the screen form;
- *Guid* – is used to identify a menu item.
  Guid is generated automatically at random and, if necessary, (in case of coincidence with Guid of another object) can be changed:

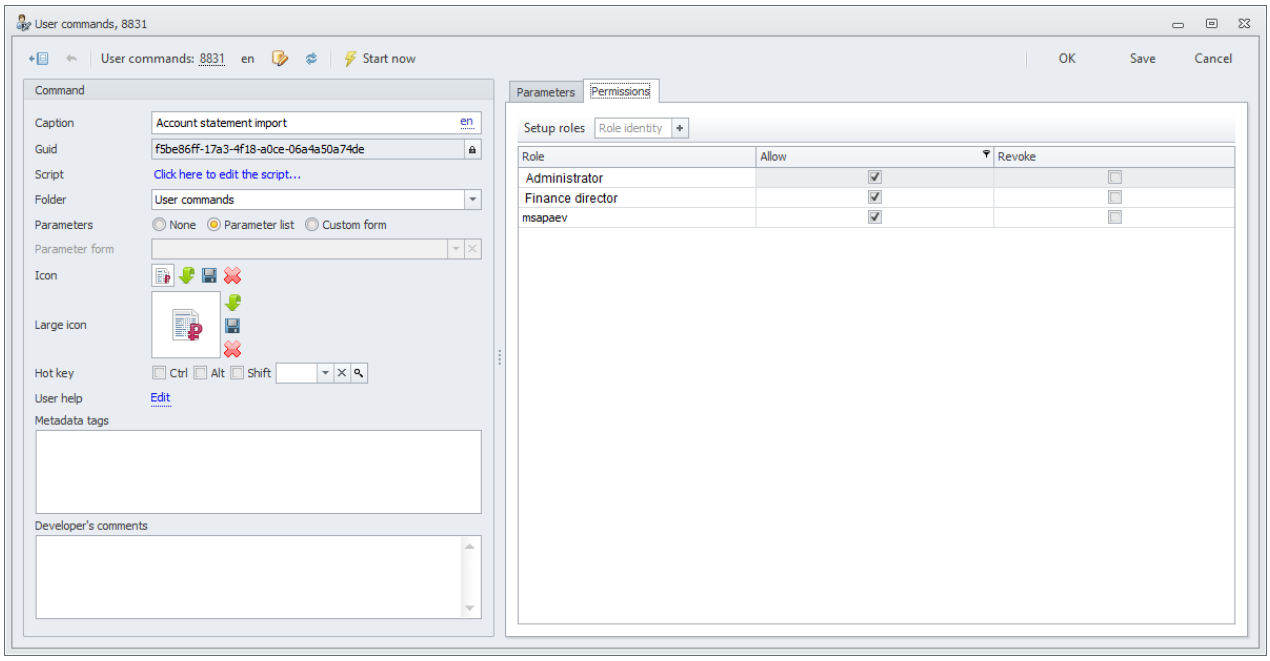


- *Script* – link to the script. In case of creation of new user command, the script is created automatically upon its saving. Click the link *Click here to edit the script...* during creation of new user command will result into saving of the command and its reloading, after that the script edit form will open;
- *Folder* – a group, the command belongs to;
- *Parameters* – application of additional parameters before execution of user command:
  - *None* – no additional parameters are requested;
  - *Parameter list* – execution of the command will be preceded with opening of the form, to be generated with Ultimate AEGIS® system, where the user is offered to fill in additional parameters, described in the list *Parameters in the right part of the command edit form:*
  - *Custom form* – execution of the command will be preceded with opening of special additional form (designed by the application developer), in which the user will be offered to fill in a number of parameters.

- *Parameter form* – a special form with additional parameters. Its selection is available if option *Custom form* is selected in item *Parameters* . This form must be preliminary designed by the application developer, e.g. in Visual Studio, and placed in the shared client module (the process is detailed in the section Request forms of parameters of interactive commands);
- *Icon* – command icon (with the size of 16 x 16 pixels).
  The buttons to the right of icon preview area allow:
  – loading the icon;
  – saving the icon previously downloaded to the computer;
  – deleting the icon;
- *Large icon* – a large icon (with the size of 32 x 32 pixels);
- *Hot Key* - keys combination which can open the user command. Using the flags, one or several functional keys (Ctrl, Alt and Shift) can be selected, and a symbol key can be selected in the control element to the right of them.
  The buttons of the control elements, using which symbol selection is made, allow:
  – to select a symbol;
  – to delete the selected value;
  – to view, if such shortcut keys occur for any other command:

  It is important to remember that user commands is invoked from the main menu, i.e. also in any open in this moment screen form, in which also may be available some specific commands. So it is very important to check hot keys combinations set for the command for conflicts. If the system has two commands, which are invokes by the same hot keys combinationby their pressingwill invoke only one of them, and it is unknown beforehand the which one.
- *User help* – a comment to the command, which the end user can see in the form of a hint, dropping down in case of mouseover at the command. The comments are entered for each of system languages in the form opened by clicking the link;
- *Metadata tags* – tags used to describe command functionality;
- *Developer's comments* – comments of the application developer;
- *Parameters* — additional parameters to use when executing the command. All these parameters are used (on the standard form) if option *Parameter list* is selected in the item *Parameter*. The parameters can be filtered by *Name* in accordance with the text entered in the field "*Name*". Each parameter has:
  - *Name* – parameter name;
  - *Caption* – name displayed in the screen forms;
  - *Type Identity* – parameter type (see details in the section Data types);
  - *Is Required* – the flag indicating if the parameter is mandatory for fill-in;
  - *Save History* – the flag indicating the need to remember the last value of the parameter entered by the user;
  - *String Size* (available for the types of data *Text* and *String*) – limits the size of parameter value by the specified value;
  - *Sort index* – index, the parameters are sorted by in the screen form. Any integer numbers can be used as index values. The parameters will be arranged in the form from top downward in the index increasing order;
  - *Default Value* (available for all data types except for *Binary*) – the default parameter value, which is used in the form of additional parameters;
  - *Referenced Dictionary ID* (available for data type *Long*)– ID of the dictionary (object), the reference to which is parameter.

The tab *Permissions* allows set up the rights to start the command or check that it is available for one role at least:



Scripts of user commands realize the interface *IUserCommand* (from namespace *Ultima.Scripting*).

The following is transferred at script input:

- additional parameters of the command (if they were requested);
- collection of actions *ClientActions,* which should be executed on the side of client application upon completion of script operation.

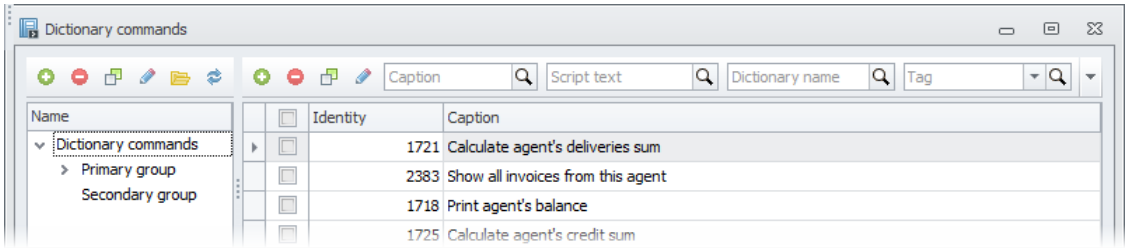
Interface *IDictionaryCommand* implements a single method,*Execute* executing the script:

- *Execute(IDictionary<string, object> parameters, IList<ClientAction> clientActions);*
  - *parameters* – command parameters;
  - *clientActions* – a collection of *ClientActions* actions.

Dictionary record commands

Document commands are the scripts, implementing *IDocumentCommand* interface and executed on the application server over dictionary record while the user selects corresponding item in the menu “Commands” in the dictionary edit form (or in the context menu, described upon a click of the right mouse button on the dictionary record in the list form). Dictionary record command can be assigned only to one dictionary. If you want to perform the same function with recordings of several directories, you should bring the overall functionality in service (see Services) and create, using this service, an individual command on the directory record for each of these directories.

A list of all dictionary record commands can be bound in the dictionary "Dictionary commands":

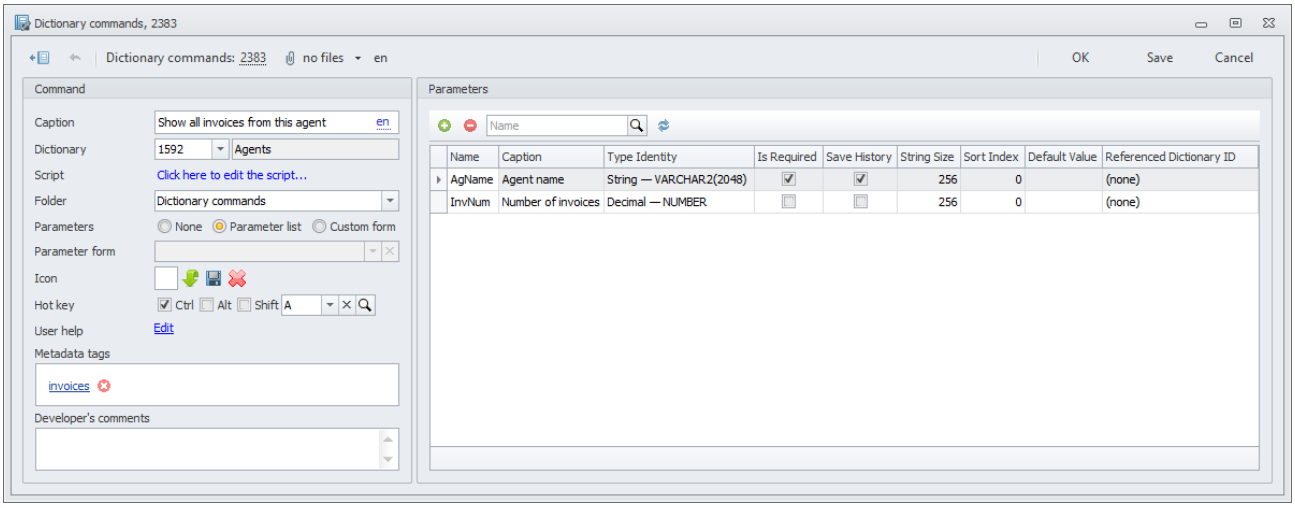


Dictionary window divided into two parts: a tree of the group of commands is displayed to the left, a list of commands selected on the left of the group is displayed to the right.

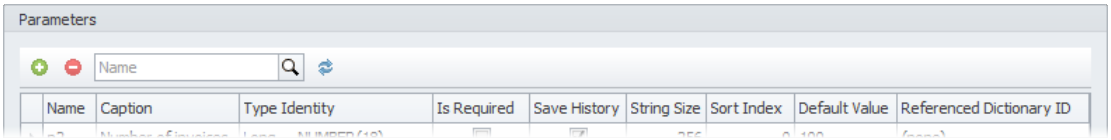
Dictionary records can be filtered by the command name displayed in the screen forms (*Caption*), *Text of its script* (*Script text*), *Name of the dictionary*, which it is associated with (*Dictionary name*) and *Tags* (*Tag*).

The script of selected dictionary record command in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

Dictionary record commands has the following properties:



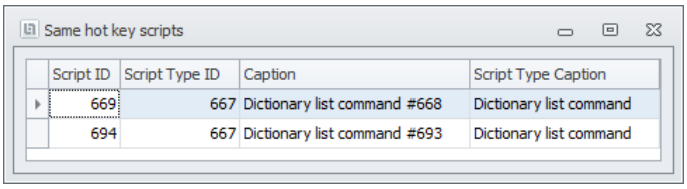
- *Caption* – command name displayed in the screen form;
- *Dictionary – a dictionary, to which records the command is applied;*
- *Script* – link to the script. In case of creation of new dictionary record command, the script is created automatically upon its saving. Click the link *Click here to edit the script...* during creation of new dictionary record command will result into saving of the command and its reloading, after that the script edit form will open;
- *Folder* – a group, the command belongs to;
- *Parameters* – application of additional parameters before execution of a dictionary record command:
  - *None* – no additional parameters are requested;
  - *Parameter list* – execution of the command will be preceded with opening of the form, to be generated with *Ultimate AEGIS®* system, where the user is offered to fill in additional parameters, described in the list *Parameters in the right part of the command edit form:*



  - *Custom form* – execution of the command will be preceded with opening of special additional form (designed by the application developer), in which the user will be offered to fill in a number of parameters.
- *Parameter form* – a special form with additional parameters. Its selection is available if option *Custom form* is selected in item *Parameters* . This form must be preliminary designed by the application developer, e.g. in Visual Studio, and placed in the shared client module (the process is detailed in the section parameters of interactive commands request form);
- *Icon* – command icon (with the size of 16 x 16 pixels).
  The buttons to the right of icon preview area allow:
  – loading the icon;
  – saving the icon previously downloaded to the computer;

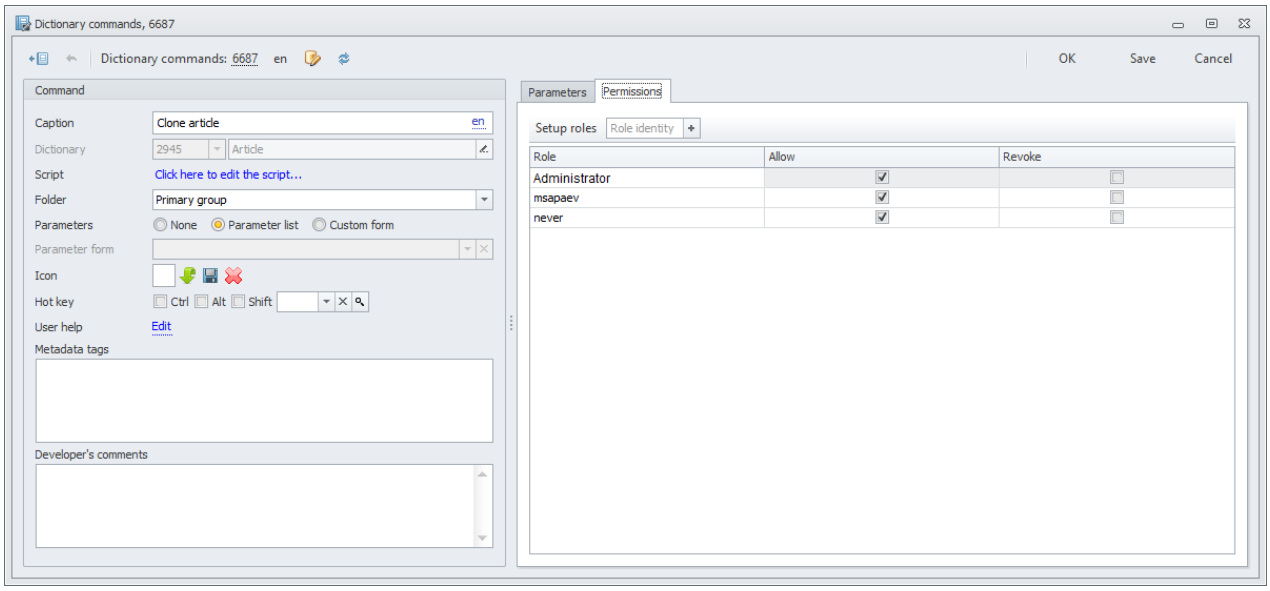
– deleting the icon;

- *Hot Key* – shortcut keys, using which a command can be called from the dictionary edit form. Using the flags, one or several functional keys (Ctrl, Alt and Shift) can be selected, and a symbol key can be selected in the control element to the right of them.
  The buttons of the control elements, using which symbol selection is made, allow:
  – to select a symbol;
  – to delete the selected value;
  – to view, if such shortcut keys occur for any other command:



- *User help* – a comment to the command, which end the user can see as a hint, dropping down in case of hovering a cursor over it. The comments are entered for each of system languages in the form opened by clicking the link;
- *Metadata tags* – tags used to describe command functionality;
- *Developer's comments* – comments of the application developer;
- *Parameters* — additional parameters to use when executing the command. All these parameters are used (on the standard form) if option *Parameter list* is selected in the item *Parameter*. The parameters can be filtered by *Name* in accordance with the text entered in the field "*Name*". Each parameter has:
  - *Name* – parameter name;
  - *Caption* – name displayed in the screen forms;
  - *Type Identity* – parameter type (see details in the section Data types);
  - *Is Required* – the flag indicating if the parameter is mandatory for fill-in;
  - *Save History* – the flag indicating the need to remember the last value of the parameter entered by the user;
  - *String Size* (available for the types of data *Text* and *String*) – limits the size of parameter value by the specified value;
  - *Sort index* – index, the parameters are sorted by in the screen form. Any integer numbers can be used as index values. The parameters will be arranged in the form from top downward in the index increasing order;
  - *Default Value* (available for all data types except for *Binary*) – the default parameter value, which is used in the form of additional parameters;
  - *Referenced Dictionary ID* (available for data type *Long*)– ID of the dictionary (object), the reference to which is parameter.

The tab *Permissions* lets set rights for starting the command or check, that at least one role is available:



Dictionary record commands scripts realize interface *IDictionaryCommand* (from namespace *Ultima.Scripting*).

The following is transferred at script input:

- dictionary record ID;
- additional parameters of the command (if they were requested);
- collection of actions *ClientActions*, which should be executed on the side of client application upon completion of script operation.

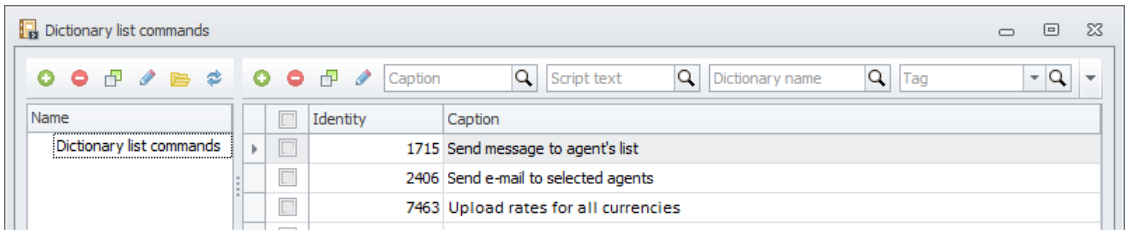
IDictionaryCommand interface implements a single *Execute* method, executing the script:

- *Execute(long recordId, IDictionary<string, object> parameters, IList<ClientAction> clientActions)*
  - *recordId* – dictionary record ID;
  - *parameters* – command parameters;
  - *clientActions* – a collection of *ClientActions* actions.

Dictionary list commands

Dictionary list commands — are scripts, realizing the interface *IDictionaryListCommand*, and which are carried out on the application server over several dictionary records, marked with flags in a list form when the user choose the corresponding item in the menu "Commands" in a list form of the dictionary. A dictionary list command can be tied to the only one dictionary. If it is required to execute identical functions with records of several dictionaries, it is necessary to take out this general functionality into the service (see the section Services) and to create, using this service, separate dictionary list commands for each of these dictionaries.

The list of all dictionary list commands can be found in the dictionary "Dictionary list commands":

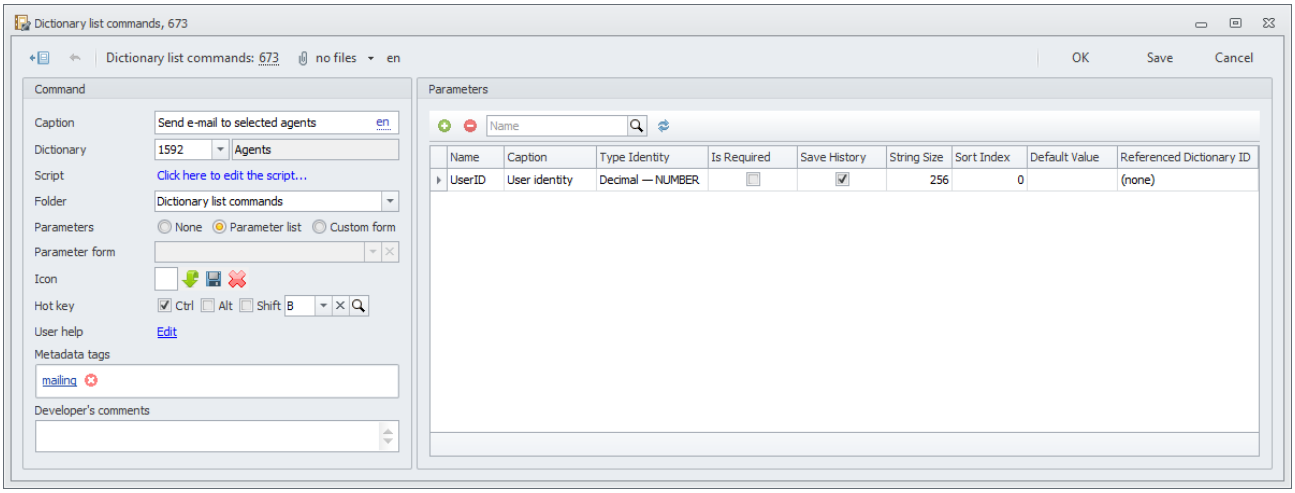


The dictionary window is divided into two parts: on the left the tree of commands groups is displayed, on the right — the list of commands of the group chosen from the left.

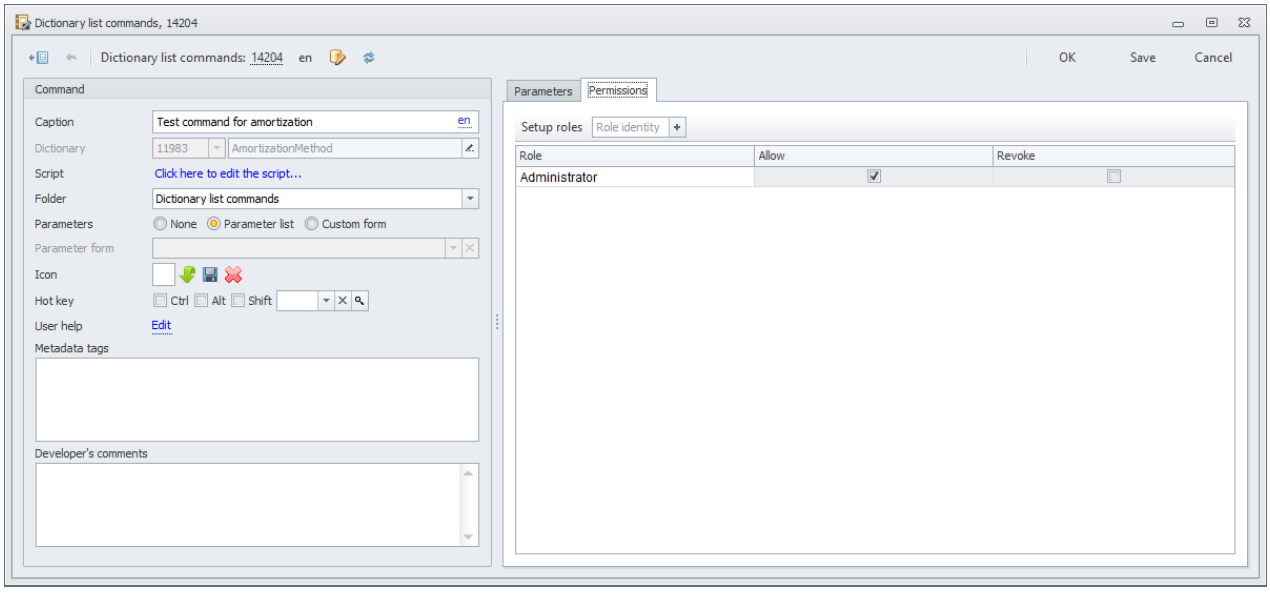
Dictionary records can be filtered by the command name *Displayed in screen forms* (*Caption*), *Text* of its script (*Script text*), *Dictionary name*, to which it is tied (*Dictionary name*) and *Tags* (*Tag*).

To open a script of the chosen dictionary list command is possible in the form of editing directly from a list form of the dictionary, chosen the item *Edit script* in a context menu.

The dictionary list command has properties completely identical to properties of the dictionary record command:



The tab *Permissions* allows quick adjusting the rights to run the command or to check that at least one role has an access to it:



Scripts of the dictionary list commands realize the *IDictionaryListCommand* interface (from the namespace *Ultima.Scripting*).

The following is transferred to an entrance of the script:

- identifiers of the dictionary records;
- additional command parameters (if they have been requested);
- Collection of actions *ClientActions*, which should be executed on the part of the client application on completion of a script work.

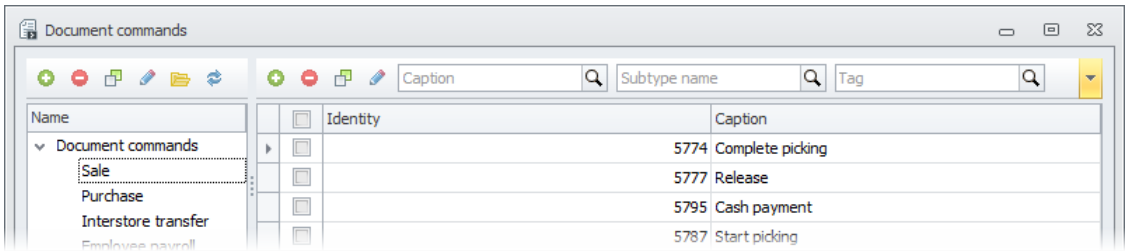
The interface *IDictionaryListCommand* implements only *Execute* method, which is carrying out the script:

- *Execute(long[] records, IDictionary<string, object> parameters, IList<ClientAction> clientActions)*
  - *records* — a list of identifiers of the dictionary records;
  - *parameters* — command parameters;
  - *clientActions* — collection of actions *ClientActions*.

Document commands

Document commands are the scripts, implementing *IDocumentCommand* interface, and executed on the application server over a document while the user selects corresponding item in the menu "Commands" in the document edit form (or in the context menu, described upon a click of the right mouse button on the document in the list form).

A list of all documents commands can be found in the dictionary "Document commands":

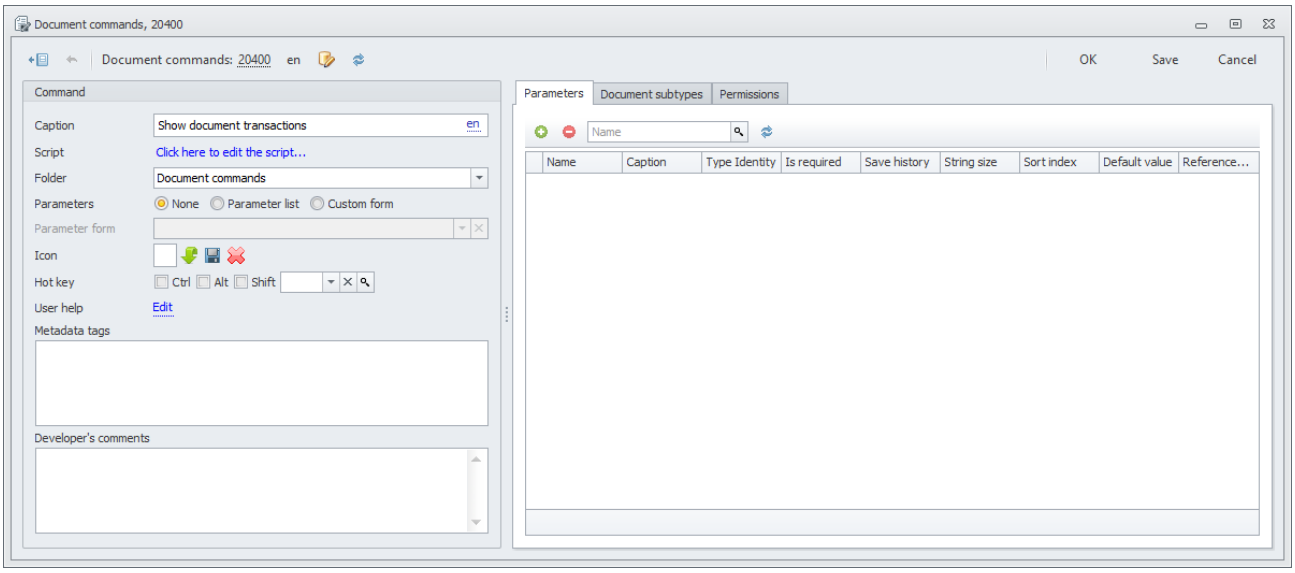


Dictionary window is divided into two parts: a tree of the commands group is displayed to the left, a list of commands of the group, selected on the left, is displayed to the right.

The dictionary records can be filtered by *the command name displayed in the screen forms* (*Caption*), *Subtype of the document*, which it is associated with (*Subtype name*) and *Tags* (*Tag*).

The script of selected document command in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

Document command has the following properties:



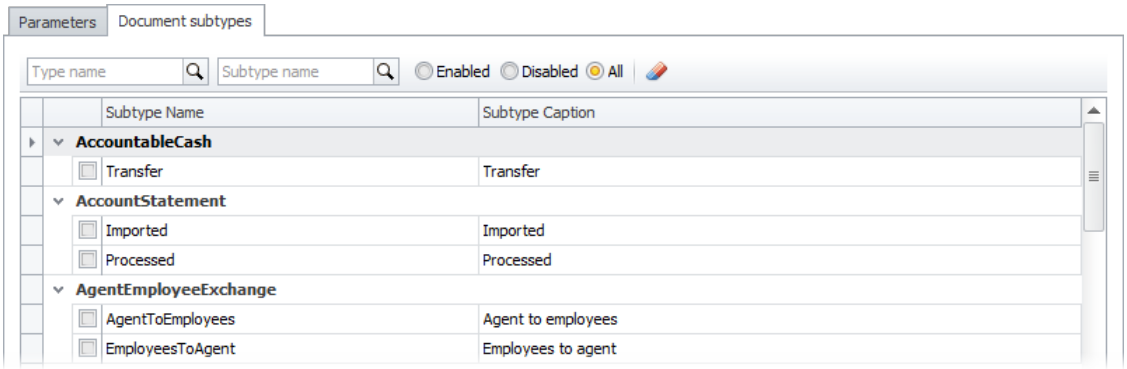
- *Caption* – command name displayed in the screen form;
- *Script* – link to the script. In case of creation of a new document command, the script is created automatically upon its saving. Click the link *Click here to edit the script...* during creation of new document command will result into saving of the command and its reloading, after that the script edit form will open;
- *Folder* – a group, the command belongs to;
- *Parameters* – application of additional parameters before execution of document command:
  - *None* – no additional parameters are requested;
  - *Parameter list* – execution of the command will be preceded by the form opening, to be generated with Ultimate AEGIS® system, where the user is offered to fill in additional parameters, described in the tab *Parameters* in the right part of the command edit form:
  - *Custom form* – execution of the command will be preceded with opening of special additional form (designed by the application developer), in which the user will be offered to fill in a number of parameters.
- *Parameter form* – a special form with additional parameters. Its selection is available if option *Custom form* is selected in item *Parameters* . This form must be preliminary designed by the application developer, e.g. in Visual Studio, and placed in the shared client module (the process is detailed in the section Request forms of parameters of interactive commands);
- *Icon* – command icon (with the size of 16 x 16 pixels).
  The buttons to the right of icon preview area allow:
  – loading the icon;
  – saving the icon previously downloaded to the computer;
  – deleting the icon;
- *Hot Key* – shortcut keys, using which a command can be called from the document edit form. Using the flags, one or several functional keys (Ctrl, Alt and Shift) can be selected, and a symbol key can be selected in the control element to the right of them.
  The buttons of the control elements, using which symbol selection is made, allow:
  – to select a symbol;
  – to delete the selected value;
  – to view, if such shortcut keys occur for any other command:
- *User help* – a comment to the command, which the end user can see in the form of a hint, dropping down in case of mouseover at the command. The comments are entered for each of system languages in the form opened by clicking the link;

- *Metadata tags* – tags used to describe command functionality;
- *Developer's comments* – comments of application developer.

In the tab «Parameters» on the right side of the edit form command additional parameters are listed that are used when it is executed. All these parameters are used (in the standard form) if option *Parameter list* is selected in the item *Parameter*. The parameters can be filtered by *Name* in accordance with the text entered in the field "*Name*". Each parameter has:

- *Name* – parameter name;
- *Caption* – name displayed in the screen forms;
- *Type Identity* – parameter type (see details in the section Data types);
- *Is Required* – the flag indicating if the parameter is mandatory for fill-in;
- *Save History* – the flag indicating the need to remember the last value of the parameter entered by the user;
- *String Size* (available for the types of data *Text* and *String*) – limits the size of parameter value by the specified value;
- *Sort index* – index, the parameters are sorted by in the screen form. Any integer numbers can be used as index values. The parameters will be arranged in the form from top downward in the index increasing order;
- *Default Value* (available for all data types except for *Binary*) – the default parameter value, which is used in the form of additional parameters;
- *Referenced Dictionary ID* (available for data type *Long*)– ID of the dictionary (object), the reference to which is parameter.

in the tab "Document subtypes", a list is given for the subtypes of documents, the command is applied to:



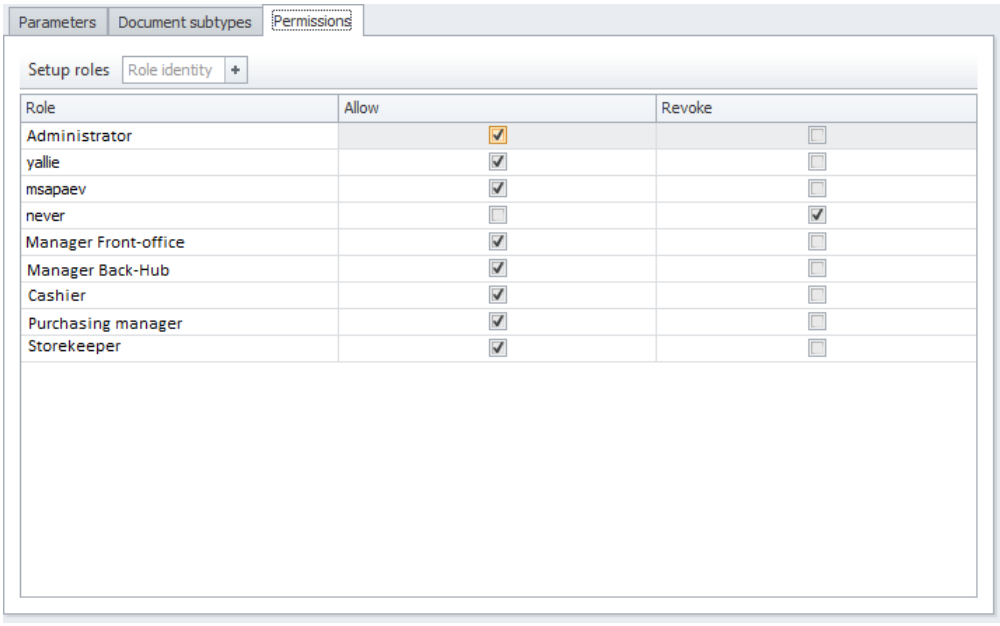
The list represents a list of all subtypes of documents grouped by the type. The command will be available from the edit form of the documents of those types, which are flagged.

The list can be filtered by *Type name* or *Document subtype name* in accordance with the text entered into the fields "*Type name*" or "*Subtype name*". The list can be additionally filtered using the flags by subtypes of documents:

- *Enable* – by all flagged subtypes;
- *Disable* – by all non-flagged subtypes;
- *All* – by all subtypes irrespective of the set flag.

A filter can be cleared and a full list of subtypes of documents can be displayed by clicking .

The tab *Permissions* lets set rights for starting the command or check, that at least one role is available:



Document commands scripts realize the interface *IDocumentCommand* (from namespace *Ultima.Scripting*).

The following is transferred at script input:

- document ID;
- additional parameters of the command (if they were requested);
- collection of actions *ClientActions*, which should be executed on the side of client application upon completion of script operation.

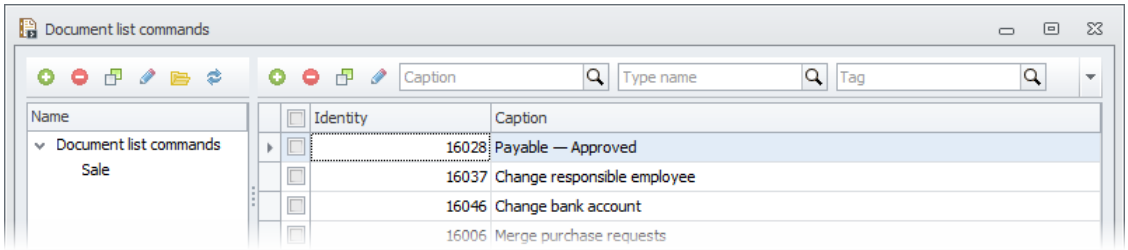
IDocumentCommand interface implements a single *Execute* method, executing the script:

- *Execute(long recordId, IDictionary<string, object> parameters, IList<ClientAction> clientActions)*
  - *recordId* – document ID;
  - *parameters* – command parameters;
  - *clientActions* – a collection of *ClientActions* actions.

Document list commands

Document list commands are scripts, realizing the interface *IDocumentListCommand;* they are carried out on the application server over several documents, marked with flags in a list form when the user choose the corresponding item in the menu “Commands” in a list form of the documents.

The list of all document list commands can be found in the dictionary "Document list commands":

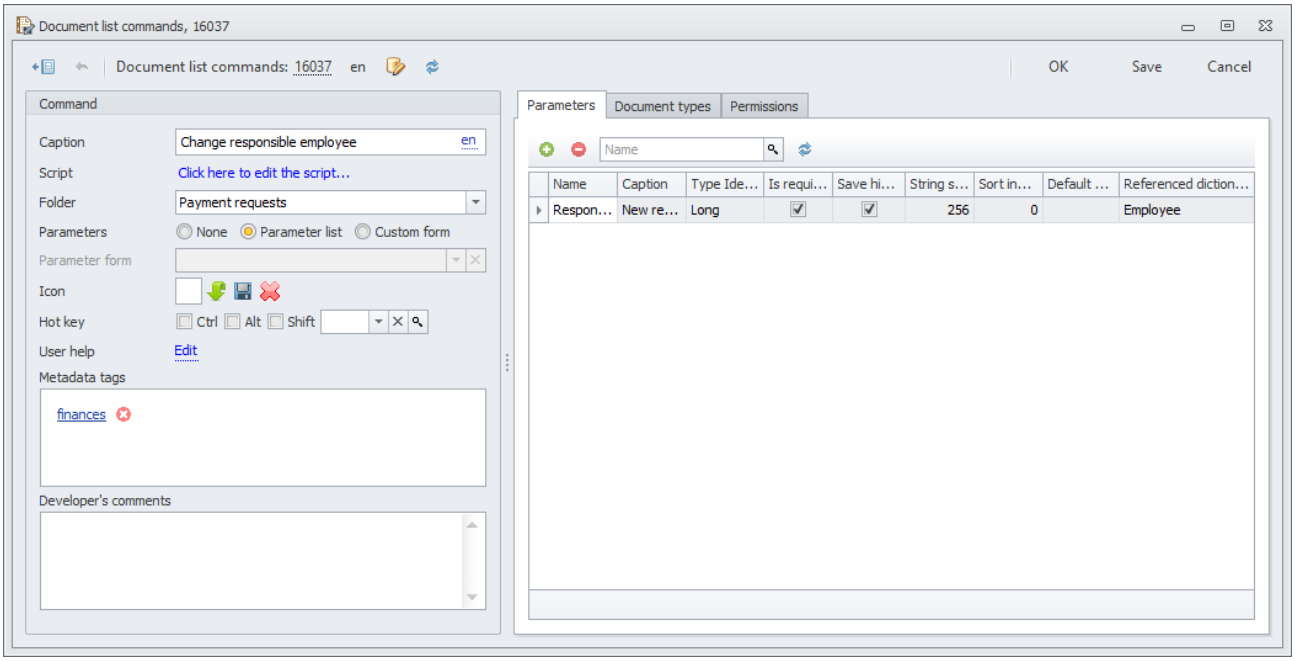


The dictionary window is divided into two parts: on the left the tree of commands groups is displayed, on the right — the list of commands of the group chosen from the left.

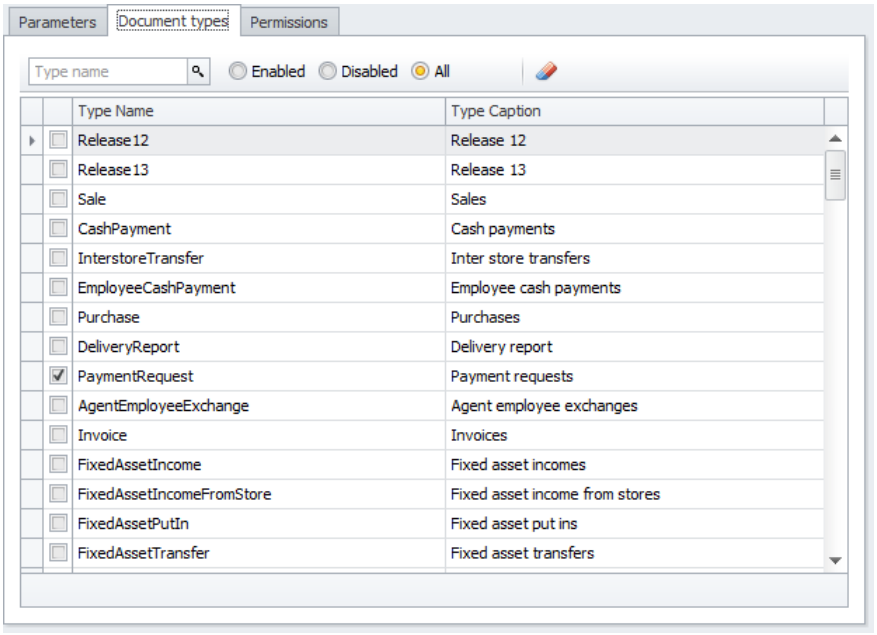
Dictionary records can be filtered by the *Displayed in screen forms name* of the command (*Caption*), *Document type*, to which it is tied (*Type name*) and *Tegs* (*Tag*).

To open a script of the chosen document command is possible in the form of editing directly from a list form of the dictionary, chosen the item *Edit script* in a context menu.

Document list command has the properties, completely identical to properties of the document command with one exception — the document list commands are applied for documents of the chosen types (but not subtypes):



In the tab «Document types» there is a list of all types of documents. The command will be available from the list form of documents marked by the flags of the types:

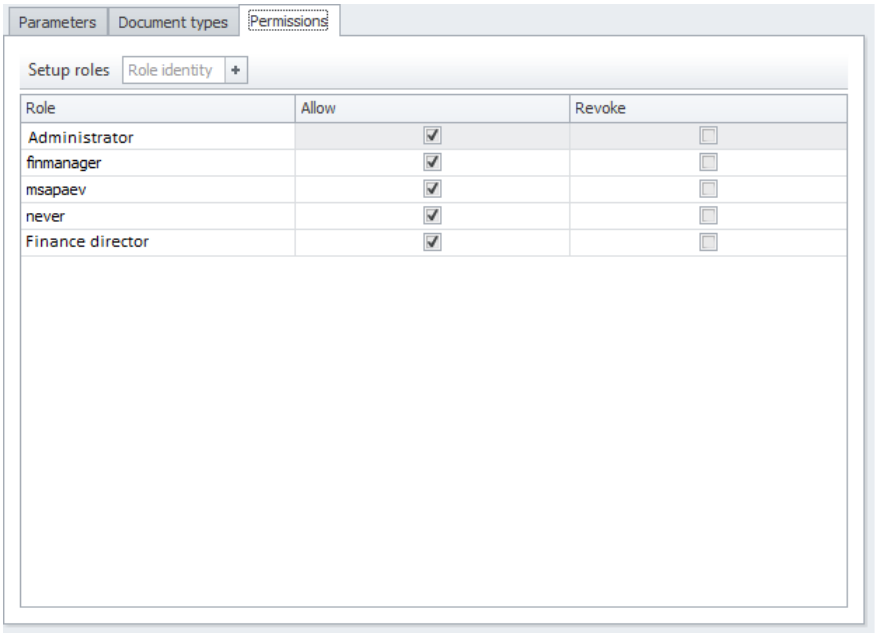


The list can be filtered by *Type name of the document* according to text added into the field "*Type name*". Also, a list can be filtered in addition by flags:

- *Enable* — in all types marked by the flags;
- *Disable* — in all types not marked by the flags;
- *All* — in all types regardless of the set flag.

To clear the contents of the filter and to display the complete list of types of the documents is possible by clicking the key button .

The tab *Permissions* allows quickly to adjust the rights to run the command or to check that at least one role has an access to it:



Scripts of the document list commands realize the *IDocumentListCommand* (from the namespace *Ultima.Scripting*).

The following is transferred to an entrance of the script:

- identifiers of documents;
- additional command parameters (if they have been requested);
- Collection of actions ClientActions, which should be executed on the part of the client application on completion of a script work.

The interface *IDocumentListCommand* implements only *Execute*, method, which is carrying out the script:

- *Execute(long[] documents, IDictionary<string, object> parameters, IList<ClientAction> clientActions)*
  - *documents* — a list of documents identifiers;
  - *parameters* — command parameters;
  - *clientActions* — collection of actions *ClientActions*.

ClientActions

ClientActions – the way of transfer from application server to client application of the information about what actions should be done on client application. You need to create the object of correspondent class and add it to the transferred collection. If the transfer of client activity collection are not foreseen in the script interface (for example, Task scripts), nobody act it.

following ClientActions (from manes space*Ultima.Client.Actions*):

- *CloseFormAction* – close present form. This action will be ignored for user command;

- *EditRecordAction* – open the record with stated ID (*RecordID*) if dictionary type (dictionaryType) for editing:

```
public class EditRecordAction : ClientAction
{
	public EditRecordAction(Type dictionaryType, long recordId)
}
```

- *MessageBoxAction* – show the window with text (*Message*) and title (*Title*):

```
public class MessageBoxAction : ClientAction
{
	public string Message { get; set; }

	public string Title { get; set; }
}
```

- *NewRecordAction* — open the form for creation a new dictionary record:

```
public class NewRecordAction : ClientAction
{
	public NewRecordAction(Type dictionaryType, IDictionary<string, object>
parameters = null)
}
```

- *NewDocumentAction* — open the form for creatin a new document:

```
public class NewDocumentAction : ClientAction
{
	public NewDocumentAction(Type documentType, IDictionary<string, object>
parameters = null)
}
```

- *OpenDocumentAction* – open document with stated ID (*DocumentID*) for editing:

```
public class OpenDocumentAction : ClientAction
{
	public OpenDocumentAction(long documentId)
}
```

- *PrintDictionaryRecordAction(Type dictionaryType, long recordId)* – open one record print dialogue for selected dictionary type:
  - *dictionaryType* – dictionary type, define the set of print form, available for selecting in print dialogue;
  - *recordId* – dictionary record ID which should be printed;
- *PrintDictionaryListAction(Type dictionaryType, long[] records)* – open the print dialogue of several records for selected dictionary type:
  - *dictionaryType* – dictionary type, define the set of print forms, available for selecting in print dialogue;
  - *records* – dictionary records IDs list, which should be printed;
- *PrintDocumentAction(long docSubtypeId, long documentId)* – open the dialogue of one document print for selected document type:
  - *docSubtypeId* – document type, identify the set of print forms, available for selecting in print dialogue;
  - *documentId* – document ID which should be printed;
- *PrintDocumentListAction(Type documentType, long[] documents)* – open the print dialogue of several documents for selected document typr:
  - *documentType* – document subtype, dentify the set of print forms, available for selecting in print dialogue;
  - *documents* – documents IDs list which should be printed;
- *PrintFormPreviewAction(long printFormId, Dictionary<string, object> parameters)* – open the form of preview of print form:
  - *printFormId* – print form ID;

- *parameters* – print form parameters;

- *PrintTextAction* – send for printing(*Text*)in code page (*CodePage*, value by default *1251*) to device (*DeviceName*). Name for printing document can becreated (*DocumentName*, value by default *Untitled*) and shoe the window of preview (*DistpalyPreview*, value by default *true*):

```
public class PrintTextAction : ClientAction
{
	public string DeviceName { get; set; }

	public string DocumentName { get; set; }

	public int CodePage { get; set; }

	public string Text { get; set; }

	public bool DisplayPreview { get; set; }
}
```

- *ReloadRecordAction* – reload current record. Both editing form and list form will be reloaded This action will be ignored by user command;
- *SaveFileAction* – save (script result) to file:

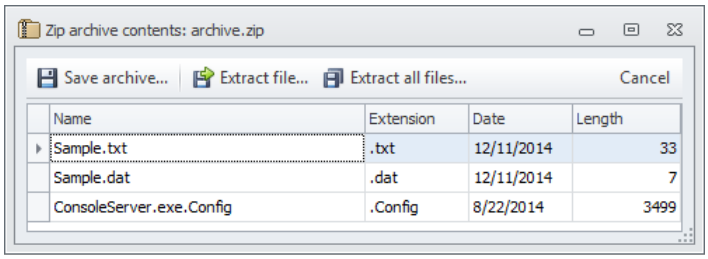
```
public class SaveFileAction : ClientAction
{
	public SaveFileAction(string fileName, DateTime? lastWriteTime = null,
		byte[] fileData, bool displayArchiveContents = false)

	public SaveFileAction(string fileName, byte[] fileData)
		: this(fileName, DateTime.Now, fileData)

	// load file for delivering to the client,
	// deleteFile parameter delete the file after loading
	public static SaveFileAction FromFile(
		string serverFilePath,
		string clientFilePath = null,
		bool deleteFile = false)
}
```

Also saving failes can be added to zip-archive:

```
// using Ultima.Client.Actions;
// using Ultima.IO.Compression;

var zip = new ZipBuilder()
	.AddText("Sample.txt", "This is a sample file. Stay awed.")
	.AddBinary("Sample.dat", new byte[] { 1,2,3,4,5,6,7 })
	.AddFile(fileName, delete: true);

clientActions.AddSaveFileAction(zip, displayArchiveContents: true);
```

- *AddText* – add text file (name and content of file in the form of lines) to the archive;
- *AddBinary* – add binary file (file name transfer in the form of line, and content as bytes massive) to the archive;
- *AddFile* – add file to the archive. If parameter *delete* set in *true*, file will be deleted from server disc after it adding to the file archive;

- *displayArchiveContents* – setting the parameter value in *true* the archive saving dialogue window is shown to the user



- *ShowExceptionAction* – show exclusion:

```
public class ShowExceptionAction : ClientAction
{
        public Exception Exception { get; set; }
}
```

- *ShowTableAction* – show table data in the form with grid title control element (*Title*, value by default – shown name of the first table from list *Tables*), accompanying text (*Info*) and settings (*SettingsSubkey*, value by default – the name of the first table from list *Tables*). Parameter *Tables* can transfer several tables, every of which will be shown in the own tab:

```
public class ShowTableAction : ClientAction
{
        public IList<SlimTable> Tables { get; private set; }

        public string SettingsSubkey { get; private set; }

        public string Title { get; set; }

        public string Info { get; private set; }
}
```

  - Switch on/off of table column is ruling by property SlimColumn.Visible.
  - Columns with underlining begins automatically mark as invisible.
  - In long type column Dictionary Type can be stated. If this property is empty, records of this dictionary will be opened by double left mouse button click on the record code cell.
  - If the Title property is empty, SlimTable.Caption property from the first table is used as the title.
  - Values of “ForeColor” and “BackColor” columns use for coloring of table lines. You can use standard colors name as Red, Maroon or LightCyan, HTML-format #A5C or #AAFF22, and even integral value ARGB. Columns with such names are hidden by form independent of Visible flag.

Example of use of API ClientActions:

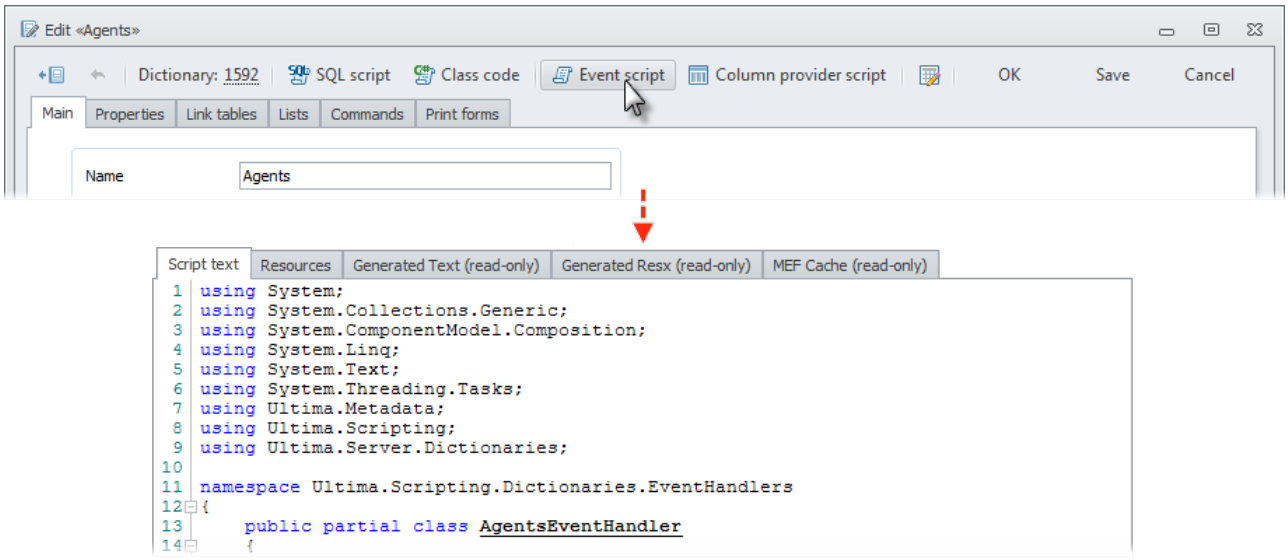
```
// using Ultima.Client.Actions;

clientActions.Add(new ReloadRecordAction());
```

Handlers

Dictionary events handlers

The dictionary events handler is a script executed automatically in case of a number of events occurring to dictionary records. Own events handler can be created for each dictionary. Click *Event script* in the dictionary edit form to create the handler. During creation of new dictionary, the events handler is not created by default.

The list of handlers of dictionary events can be found in the dictionary "Scripts". Besides, the handler of events of particular dictionary can be opened from its edit form:



The handlers of dictionary events implement *IDictionaryEventHandler* interface.

The handler can respond to the following events, which occur to dictionary records:

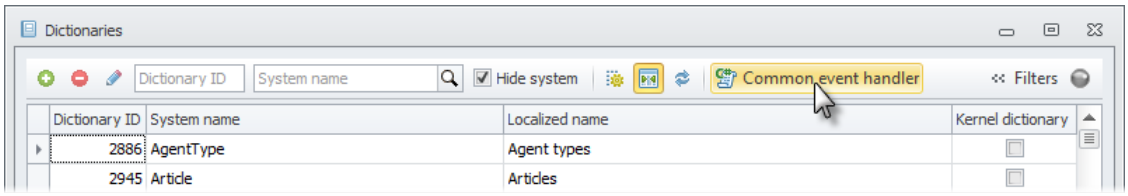
- *BeforeCreate* – the handler is executed directly before creation of dictionary record, after the user has clicked a button for creation of new record.
  - the following is delivered at handler input:
    - a dictionary record being saved;
    - parameters of saving;
- *BeforeClone* – the handler is executed before the cloning of a dictionary record. It helps to prepare the record for cloning: clear unnecessary data, replace the record name to remove duplicates, etc.
  - the following is delivered at handler input:
    - the original dictionary record, which is to be cloned;
    - additional saving parameters for the new record;
- *AfterLoad* – the handler is executed after opening the dictionary record, but is not executed in case of new record creation. Using the handler, e.g. additional parameters can be loaded into the dictionary record.
  - the following is delivered at handler input:
    - a dictionary record being opened;
    - a flag defining if internal objects of dictionary record will be loaded. The internal objects of dictionary record are for instance its properties-links. In case of indication of the need in loading of internal objects as the values for such properties-links, the records will be loaded, which they refer to, in total. Otherwise, only IDs of the records, which the properties-links refer to, will be loaded;
- *BeforeSave* – the handler is executed directly before saving the dictionary record, after the user has clicked save button. Using the handler, e.g. the data can be checked and, if necessary, modified before saving.
  - the saved dictionary record is delivered at handler input;
- *AfterSave* – the handler is executed after saving of dictionary record, including after execution of BeforeSave handler, but before transaction commitment. Using the handler, saving can be canceled, having thrown an exception, and modifications made to the dictionary record can be canceled (changes cannot be made to already saved dictionary record).
  - the saved dictionary record is delivered at handler input;
- *SaveFailed* – the handler is executed in case of throwing an exception during operation for saving of the dictionary record.

the following is delivered at handler input:
- a dictionary record being saved;
- an exception, thrown during saving;

- *BeforeDelete* – the handler is executed directly before deletion of dictionary record, after the user has clicked deletion button. Using the handler, deletion process can be canceled having thrown an exception.
  the ID of deleted dictionary record is delivered at handler input;
- *AfterDelete* – the handler is executed after deletion of the dictionary record (the dictionary record does not exist any more), including after execution of *BeforeDelete* handler, but before transaction commitment. Using the handler, deletion can be canceled having thrown an exception.
  the ID of deleted dictionary record is delivered at handler input;
- *DeleteFailed* – the handler is executed in case of throwing an exception during operation for deletion of the dictionary record.
  the following is delivered at handler input:
  - ID of dictionary record being deleted;
  - an exception, thrown during deletion.

In addition to the handler of events of particular dictionary (which may be even not created), there is the common handler of dictionary events *CommonDictionaryEventHandler*, which responds to the events, which occur to the records of any dictionary. It responds to the same events but is executed always before the handler of events of particular dictionary.

The handler of events *CommonDictionaryEventHandler* can be found in the dictionary "Scripts" or opened via the list form of the dictionary Dictionaries:



The sequence of calls of dictionary events handlers:

1. In case of saving:
   - *BeforeSave* event *CommonDictionaryEventHandler* handler;
   - *BeforeSave* dictionary's event handler;
   - saving of the record changes in DB;
   - *AfterSave* event *CommonDictionaryEventHandler* handler;
   - *AfterSave* dictionary's event handler;

   If any error occurs at any stage of saving, the *SaveFailed* event handler will be called (at first *CommonDictionaryEventHandler*, then the dictionary).
2. In case of deletion:
   - *BeforeDelete* event *CommonDictionaryEventHandler* handler;
   - *BeforeDelete* dictionary's event handler;
   - deletion of the record in DB;
   - *AfterDelete* event *CommonDictionaryEventHandler* handler;
   - *AfterDelete* dictionary's event handler;

   If any error occurs at any stage of deletion, *DeleteFailed* event handler will be called (at first *CommonDictionaryEventHandler*, then the dictionary).
3. In case of creation:
   - creation of the object of corresponding class;
   - receipt of new ID;
   - processing of transferred parameters (parameters, which name coincide with the dictionary fields, are assigned to dictionary fields automatically);

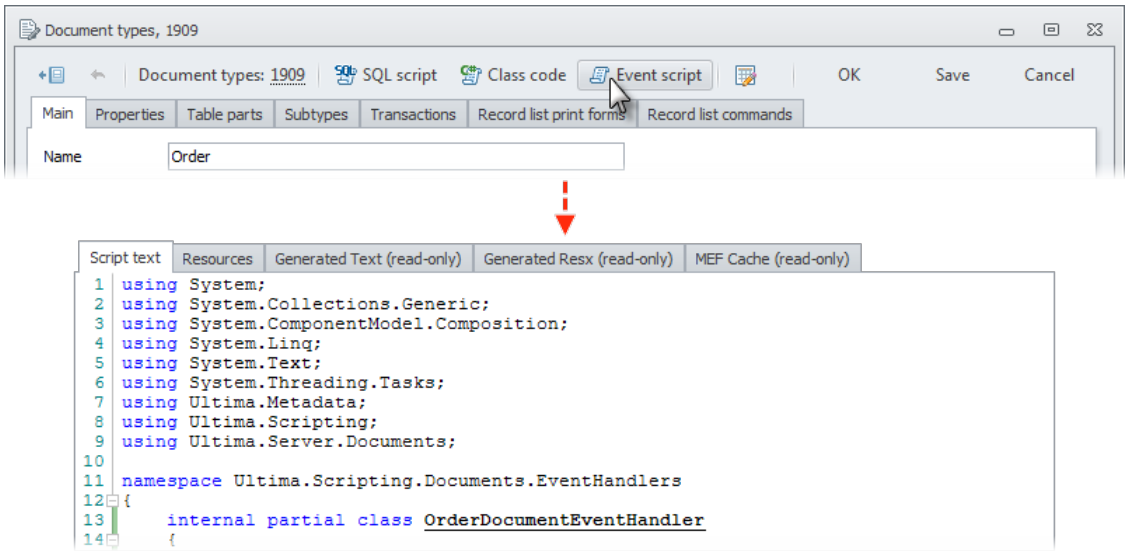
- *BeforeCreate* event *CommonDictionaryEventHandler* handler;
- *BeforeCreate* dictionary's event handler.

4. In case of record loading:
   - loading of the record from DB and initiation of the object;
   - *AfterLoad* event *CommonDictionaryEventHandler* handler;
   - *AfterLoad* dictionary's event handler.

Document events handlers

The total events handler is a script executed automatically in case of a number of events occurring to the total. Own events handler can be created for each dictionary. Click *Event script* in the document type edit form to create the handler. During creation of new document type, the events handler is not created by default.

The list of handlers of documents events can be found in the dictionary "Scripts". Besides, the handler of events of particular dictionary can be opened from its edit form:



The handlers of document events implement *IDocumentEventHandler* interface.

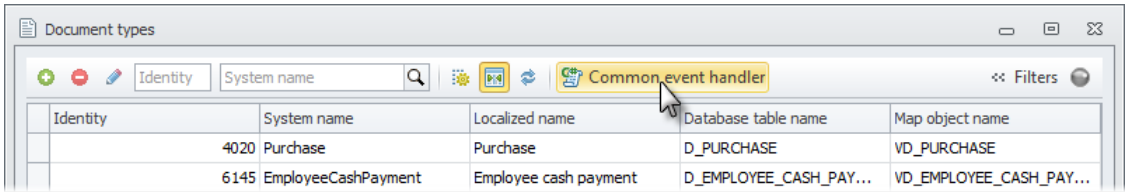
The handler can respond to the following events, which occur to dictionary records:

- *BeforeCreate* – the handler is executed directly before creation of document, after the user has clicked a button for creation of new record. If the document has several original subtypes, execution of handler will precede opening subtype selecting form.
  the following is delivered at handler input:
  - Creating document;
  - Creation parameters;
- *BeforeClone* – handler executed before document cloning. It let to prepare document for cloning: Clear unnecessary table parts, clear fields, original subtypes and etc.
  the following is delivered at handler input:
  - Original document for cloning
  - Additional parameters of document creation, including subtype code;
- *AfterLoad* – the handler is executed directly before creation of document, after the user has clicked a button for creation of new record. Using the handler, e.g. additional parameters can be loaded into the document.
  he following is delivered at handler input:
  - Opening document

    - a flag defining if internal objects of document will be loaded. The internal objects of document are for instance its properties-links. In case of indication of the need in loading of internal objects as the values for such properties-links, the records will be loaded, which they refer to, in total. Otherwise, only IDs of the records, which the properties-links refer to, will be loaded;
- *BeforeSave* – the handler is executed directly before saving dictionary record, after the user has clicked save button. Using the handler, e.g. the data can be checked and, if necessary, modified before saving.
  ↦ the saved document is delivered at handler input;
- *AfterSave* – the handler is executed after saving of document, including after execution of BeforeSave handler, but before transaction commitment. Using the handler of this event, saving can be canceled, having thrown an exception, and modifications made to the document can be canceled (modifications cannot be made to already saved document).
  ↦ the saved document is delivered at handler input;
- *SaveFailed* – the handler is executed in case of throwing an exception during operation for saving of the document.
  ↦ the following is delivered at handler input:
    - Saving document;
    - an exception, thrown during saving;
- *GenerateDescription* – handler executed after document saving , but before event handler*AfterSave*. Using the handler you can generate document description (value of field *Description*), which is generated by pattern by default *{DocumentType}({DocumentSubtype}) #{ID} {TRANSACTION_DATE}*.
  ↦ the following is delivered at handler input:
    - Saving document;
    - Description value;
- *BeforeDelete* – the handler is executed directly before deletion of document, after the user has clicked deletion button. Using the handler, deletion process can be canceled having thrown an exception.
  ↦ the ID of deleted document is delivered at handler input;
- *AfterDelete* – the handler is executed after deletion of the *dictionary record (the dictionary record does not exist any more), including after execution of BeforeDelete* handler, but before transaction commitment. Using the handler, deletion can be canceled having thrown an exception.
  ↦ the ID of deleted document is delivered at handler input;
- *DeleteFailed* – the handler is executed in case of throwing an exception during operation for deletion of the document.
  ↦ the following is delivered at handler input:
    - id – ID of the document being delete;
    - an exception, thrown during deletion.

In addition to the handler of events of document type (which may be even not created), there is the common handler of dictionary events *CommonDocumentEventHandler*, which responds to the events, which occur to the records of any document. It responds to the same events but is executed always before the handler of events of document type

The handler of events *CommonDocumentEventHandler* can be found in the dictionary “Scripts” or opened via the list form of the dictionary Document types:



The sequence of calls of document events handler is as follows:

1. In case of saving:
    - *BeforeSave event handler CommonDocumentEventHandler*;

- Event handler *BeforeSave* of document;
- Transaction scripts;
- Transaction validators;
- saving of the head and data of table parts in DB;
- saving of transactions in totals;
- Event handler *GenerateDescription* of document;
- handler of event *AfterSave CommonDocumentEventHandler*;
- *AfterSave* event handler of the document;

If any error occurs at any stage of saving, *SaveFailed* event handler (at first *CommonDocumentEventHandler, then the document).*

2. In case of deletion:
   - *BeforeDelete event handler CommonDocumentEventHandler*;
   - *BeforeDelete* event handler Common*Document*EventHandler;
   - deletion of the document in DB;
   - deletion of transactions from totals;
   - *AfterDelete event handler CommonDocumentEventHandler*;
   - *AfterDelete* event handler Common*Document*EventHandler;

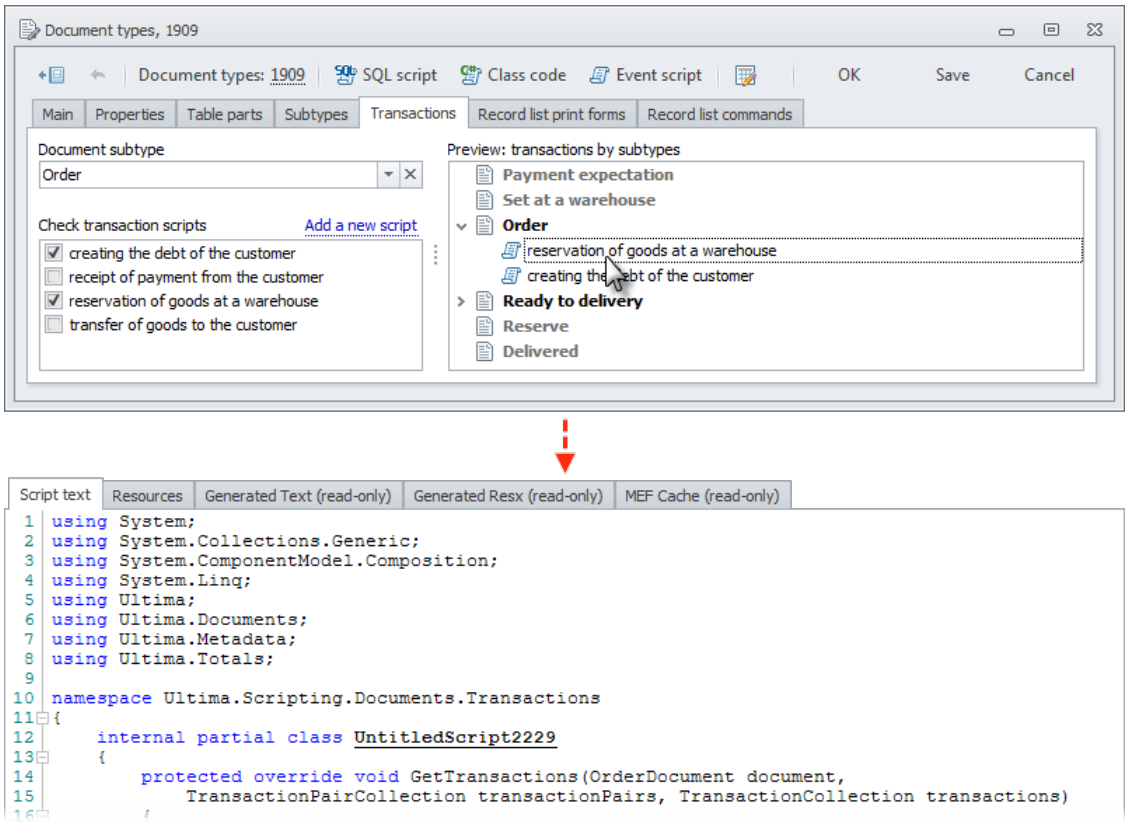
   If any error occurs at any stage of deletion, *DeleteFailed* event handler will be called (at first *CommonDocumentEventHandler, then the document).*
3. In case of creation:
   - creation of the object of corresponding class;
   - receipt of new ID;
   - Receipt of stated subtype;
   - processing of transferred parameters (parameters, which name coincide with the document fields, are assigned to document fields automatically);
   - *BeforeCreate* event handler *CommonDocumentEventHandler*;
   - *BeforeCreate event handler* of the document
4. In case of document loading:
   - loading of the document from DB and initiation of the object;
   - *AfterLoad* event handler *CommonDocumentEventHandler*;
   - *AfterLoad event handler* of the document.

Transaction scripts

The transaction scripts are the scripts forming the transactions over totals.

The transaction scripts are associated with document subtype and executed during saving of the document of particular subtype.

The list of all transaction scripts can be found in the dictionary "Scripts". Besides, the transaction scripts can be opened (as well as created, deleted or renamed) in the document type edit forms, corresponding to its subtype:



The transaction scripts are derived from *DocumentTransactionScriptBase<T> class* (*T* – document type), which implements in turn *IDocumentTransactionScript<T> interface*.

The following is transferred at script input:

- saved document;
- collection of the pair of transactions (for balance totals);
- collection of the transactions (for non-balance totals).

The transaction scripts are called with the kernel at each document saving (after successful execution of the handler before document saving). Several transaction scripts forming the transaction over various totals can be associated with each document subtype. Moreover, it should be remembered that the sequence of their call cannot be predicted.

As a result of successful execution, the transaction script returns an array of transactions, and the kernel will call the handler after document saving.

The kernel executes the minimum required set of queries to bring into compliance the set of transactions in the base and set of generated transaction scripts of transactions. As a result:

- if the base contains no transactions generated with transaction scripts, they will be added;
- if the base contains the transactions, which are missing in the set, generated with transaction scripts, they will be deleted;
- etc.

Correspondingly, if the transaction must be stored in the database throughout entire document life time in case of change of its subtype, the transaction script generating this transaction must be linked to all document subtypes.

The transactions contain a date of document transaction. Correspondingly, change of the document transaction date, even in case of no other changes, results into change of the whole set of transactions. Generally, it does not result into locks during record of transaction in the total, but increases considerably the time for transaction execution. It is recommended to avoid excessive changes of document transaction date.

The transaction scripts are designed to form an array of transaction over totals based on the document. Performance of any other manipulations with the documents is extremely undesirable using them. The event handlers are designed for that.

Peculiarities of recording transactions

The totals can be ***balance and non-balance***.

In case of changes made to the balance totals, a double-entry rule is always applied when each transaction has paired transaction (opposite in sign) and their amount is always zero.

The transaction for a transaction by the total and is called full if the values (not equal to null) are indicated for all its dimensions and variables (see details in the section Transaction scripts).

Posting is a pair of transactions for balance totals. Sometimes the term posting is applied for several pairs of transactions. For non-balance totals, the posting and transaction are synonyms.

Operational total – a total, which all dimensions and variables are set at the moment of document transaction (see details in the section Totals). All non-balance totals are operational. All balance totals are non-operational – ***analytical***.

Turnover total – a total, in which the amount balances are accumulated, and the number is always zero. The examples of such totals can be *Sale* and *Convertation*.

The total, which stores information about quantitative indicators of monetary units in particular currency, is called monetary. Moreover, it must have:

- dimension (operational) *Currency* –currency;
- variable (operational) *CurrencyAmount* – amount in currency;
- variable (analytical) *Amount* – amount in reference currency (ruble for Russia), prime cost.

Money non-negotiable total, at which *CurrencyAmount* may gain negative values, is called ***cashless***. All other money non-negotiable totals – ***cash*** . For calculation of their analytical variables, corresponding drivers of the totals are used: *CashlessMoneyTotalDriver* for cashless totals and *MoneyTotalDriver* for cash.

Money negotiable total – *Convertation*.

All transactions for money totals are made through intermediate total of *Convertation*. In the basic configuration, checking of this rule is carried out using scripts-transaction validators on money totals (see details in the section Transaction validators). In own configuration, the application developer is responsible for observance of this rule.

The money totals differ from common ones with the ability to get into arrears. From business logic perspective, it means overdraft or debt. Getting into arrears is usually possibly only for the totals, which store information about cashless money, such as bank accounts. Though in practice, it is recommended to use convertation while handling all money totals, even if cash money is taken into consideration - just not to think once again if to use convertation or not.

The ability to go into arrears means that the total can calculate the prime cost not only for outcome but for income transactions (only if uncleared overdraft is present on money total). It is made so that during

clearing off of overdraft to zero, the prime cost of zero balance on the total appears to be zero too. If doing direct transactions to such total from operational totals, by passing convertation, violation of the double-entry rule can be produced after calculation of full transactions of totals by calculator. It occurs as follows:

- for instance, if there is direct transaction from the contractors' debt to the balances of settlement accounts. If the settlement account is in arrears, it will calculate the prime cost of arrived currency. However, another prime cost will be indicated in the pair transaction since all variables are operational in total of the contractors' debt and shall not be recalculated. As a result not the right amount, which will be entered on the account, will be written off. Calculation of totals will register a critical error;
- similar situation occurs when direct transaction is made from one account to another. If the receiving account is in the overdraft, both totals will calculate a prime cost of money, each in own manner. As a result, violation of the double-entry rule will occur and, as a result, a critical error of calculation of totals.

The specified problem is solved with the use of intermediate convertation total. Instead of direct transaction *debt -> account* we make a pair of transactions: *debt -> convertation*, *convertation -> account*. In the first transaction, the amounts are indicated (since the total of debts is operational, the amount cannot be indicated), in the second transaction they are not indicated. The second transaction will be calculated with the totals' calculator. Moreover, the currency prime cost will determine the receiving total if in arrears or otherwise convertation total. As a result, all discrepancies in the amounts will remain on the convertation total, and the double-entry rule will not be violated with either of the transactions.

Therefore, the convertation total acts as buffer, on which all income and losses will fall, being a result of currency exchange transactions.

In order to prevent gross errors in the transactions by money totals, scripts validating transactions are present in the basic configuration. The validators cancel transaction of the document if the money transactions do not pass the convertation total. If during improvement of the configuration by the application developer a new money total was added, it should be taken care to write similar validating script for this total.

Handlers of total events

The total events handler is a script executed automatically in case of a number of events occurring to the total. Own events handler can be created for each total. To create the handler it is necessary to click *Event script* button in the total edit form. During creation of new total, the events handler is not created by default.

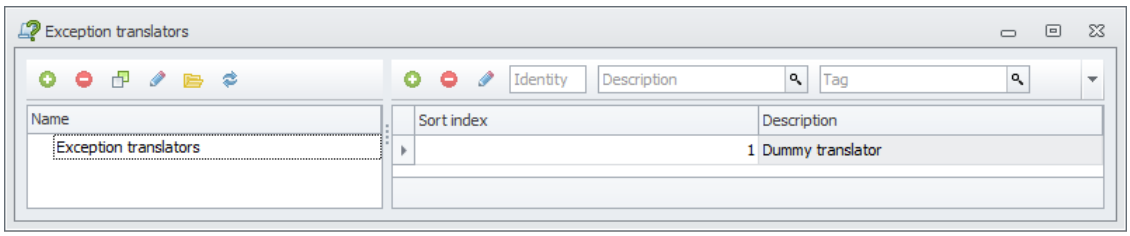
The list of all handlers of totals events can be found in the dictionary "Scripts". Besides, the handler of events of particular total can be opened from its edit form:

Exception translators

Exception translators are simple services with the *Translate method* which accept exception and return translating result. If it is impossible to return the translating result, they return *null*. User translators are executed after system translators, but before exception translators. For example, a translator which by constraint name reports object name and property belongs to system translators: it is impossible to delete record as a dictionary, a field refer to it (if the object isn't described in meta data, in the message there will be a name of the table and a column).

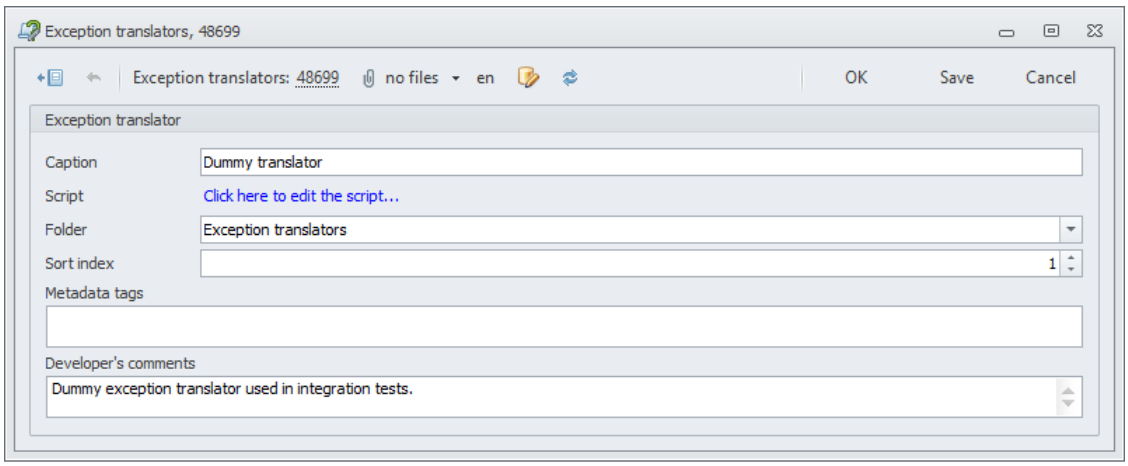
From exception translators translators are distinguished by use of scripts that gives more flexible approach to exception handling, but does them a little more difficult in application.

The list of all user exception translators can be found in the Exception translators dictionary:



The dictionary records can be filtered by *Translator name* (*Description*) and *Tags* (*Tag*).

The translator exception has the following properties:



- *Caption* – translator name;
- *Script* – a link to translator script. In case of creation of new translator, the script is created automatically upon its saving. Click on the link *Click here to edit the script...* during creation of new translator will result its saving and reloading, after that the script edit form will open;
- *Folder* – a group, the translator belongs to;
- *Sort index* – a sorting index which defines execution order of translators. Translator with the smallest index value is executed at first and further according to index value increase before the first successful execution (if it is executed successfully in the translator, reverse it isn't specified);
- *Metadata tags* – tags used to describe report translator;
- *Developer's comments* – comments of application developer.

Example of translator:

```
namespace Ultima.Scripting.Exceptions
{
	public Exception Translate(Exception ex)
	{
		//Condition check
		if (ex.WithInnerExceptions().Any(x => x.Message == "Untranslated"))
		{
			//check of serializability of original exception
			if (!ex.IsSafelySerializable())
				{
					ex = ex.ToSerializable();
				}

			//resetting of translating exception
			return new UltimaException("Translated", ex);
```

```
            }

            //the exception doesn't meet the checked conditions
            return null;
        }

        public bool StopOnSuccess
        {
            //not to execute the following translator if this worked successfully
            get { return true; }
        }
}
```

Analytic columns providers

In dictionaries, documents and table parts of document it is often required to display additional information that is not stored directly in their tables. For example, columns with the translations of the names into supported by the system languages can be added to metadata dictionaries. In the currency dictionary it is convenient to bring the column of the current exchange rate of each currency against ruble, and by double-click on this column to open the table with other available courses. In article table part, you can show your current balance for each article in stock, and so on.

The columns, in which this additional information is shown, are called *analytic columns*. Uploading information into these columns is performed by a particular type of handlers, called *analytic columns providers*. You can add analytical columns into dictionaries and documents registers with the help of analytic columns selection, in table parts of document, these columns are always displayed if AutoPopulateGridColumns flag is set to true.

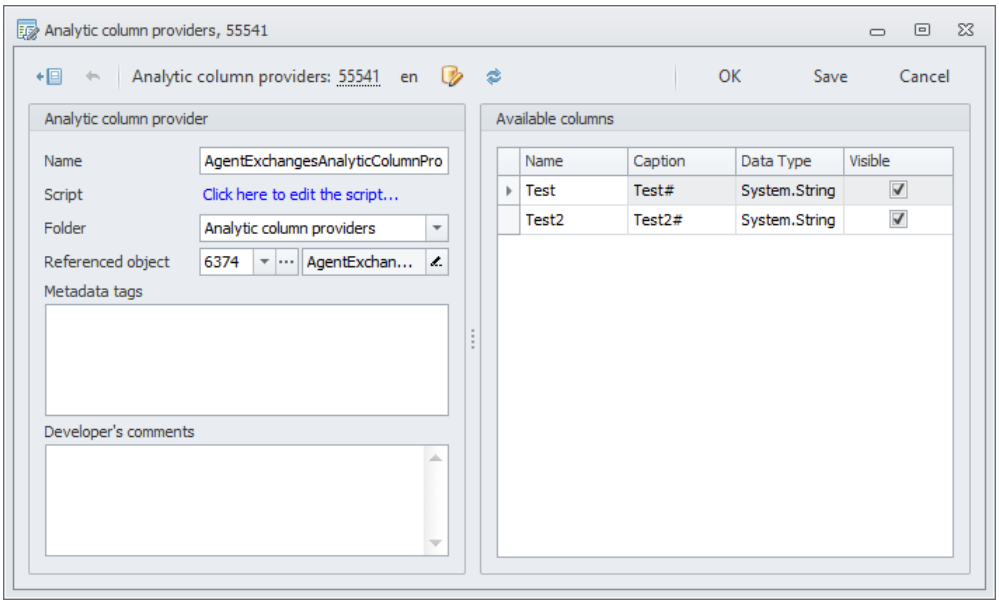
Analytic columns provider perform the following tasks:

- Provides a list of supported columns in a SlimColumn collection (for the selection of columns);
- Loads data for the selected columns and returns them as SlimTable table;
- Processes double-click on the analytical column in the line of the list and returns client actions list.

Most often, the provider takes data from other applicatiin tables, but its possibilities are not limited to this. To download the data it can connect to a different database or, let's say, refer to the Web service. Processing of double-click on the analytical column makes it easy to implement functionality of type "view details" (drill down): for example, by double-click on a column of article stock in central store a table of stock in any other stores is loaded.

Provider distinguishes the columns by their names (Name). Columns names must be unique to each provider (the system is able to distinguish between columns of the same name, received from different providers). The displayed name of columns should be localized, for that, as usual, scripting resources are used.

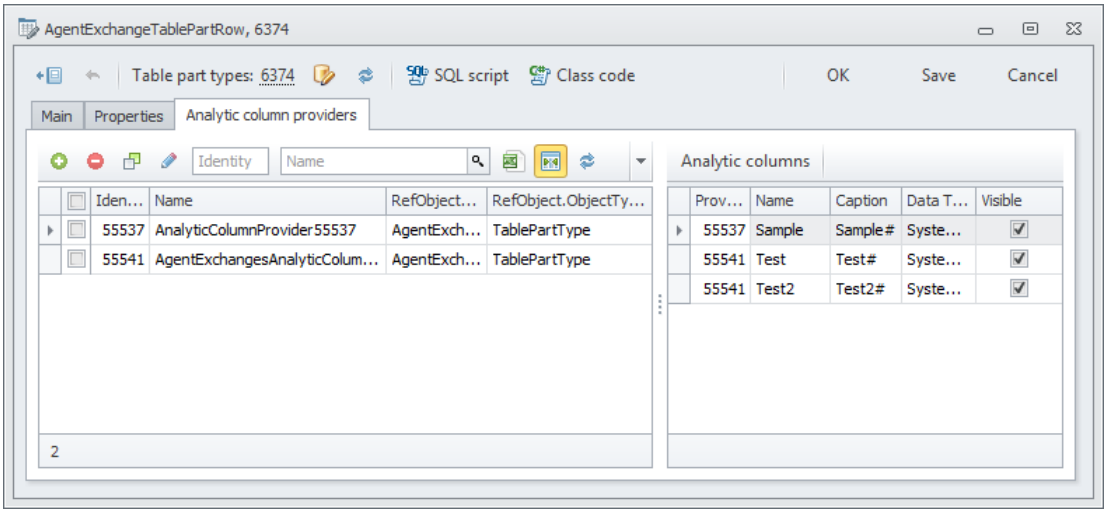
Analytic columns provider has the following features:



- *Name* – provider class name.
- *Script* – a link to provider script. The new provider of columns script is created automatically when you save. Click on the link *Click here to edit the script...* during creation of new provider will result into saving of the provider and its reloading, after that the script edit form will open;
- *Folder* – a group, the provider belongs to;
- *Referenced object* — metadata object for which the provider loads the additional column;
- *Metadata tags* – tags used to describe provider functionality;
- *Developer's comments* – comments of application developer.

The right side of the edit form displays a list of of columns, given by the provider (if the script is saved and compiled with no errors). Displayed names of of columns in the list are displayed in the language of the current user.

In metadata objects associated providers of columns are displayed in a separate tab. Here, the left side shows a list of all the providers linked to the project, and the right the list of of columns of selected provider (or all providers, if no provider is selected on the left):



The provider of analytical of columns is obliged implement three methods:

- *GetAvailableColumns* — returns columns for selection (SlimColumn list);
- *GetColumnsValues* — loads the data of columns for the list of records (SlimTable);
- *ExecuteAction* — processes mouse double-click on a column and returns the result (ClientAction list).

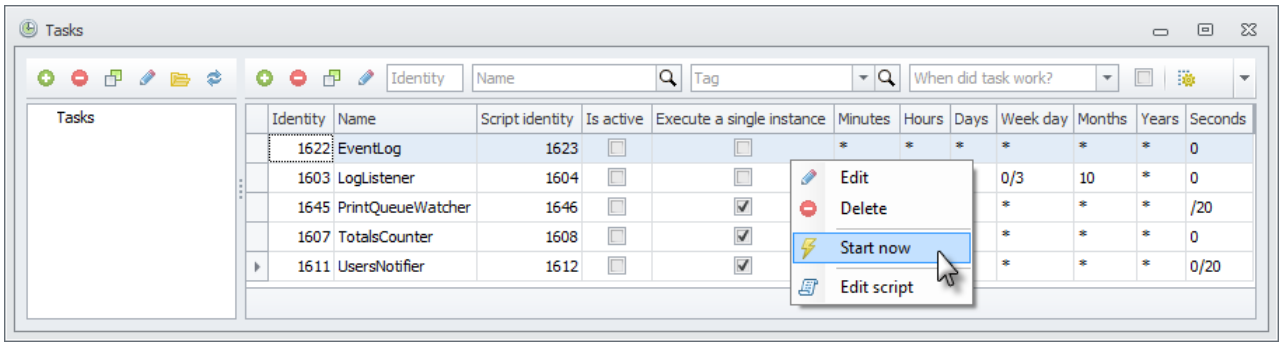
Example of script provider of analytical columns:

```
internal partial class SampleColumnProvider
{
    public IList<SlimColumn> GetAvailableColumns(Type type)
    {
        // return the list of available columns
        var columns = new List<SlimColumn>
        {
            new SlimColumn
            {
                Name = "Sample",
                Caption = "Sample#", // use resources to localize captions
                DataType = typeof(string)
            }
        };

        return columns;
    }

    public SlimTable GetColumnsValues(Type type, List<SlimColumn> columns, IDList records)
    {
        var result = new SlimTable();
        result.Columns.Add("ID", typeof(long));
        foreach (var col in columns)
        {
            result.Columns.Add(col);
        }

        foreach (var id in records)
        {
            var row = result.NewRow();
            row["ID"] = id;

            foreach (var col in columns)
            {
                if (col.Name == "Sample")
                {
                    row[col.Name] = "Sample#" + id % 100;
                }
            }

            result.Add(row);
        }

        return result;
    }

    public void ExecuteAction(Type objectType, long id, SlimColumn column, object value,
IList<ClientAction> clientActions)
    {
        clientActions.AddMessageBoxAction(new { id, value }.ToString(), column.Caption);
    }
```

```
}
```

Tasks

 Tasks are scripts performed on an application server according to schedule. A list of all tasks can be found in the "Tasks" dictionary:



The dictionary window is divided into two parts: on the left, there is a tree of task groups; on the right - a list of tasks selected to the left.

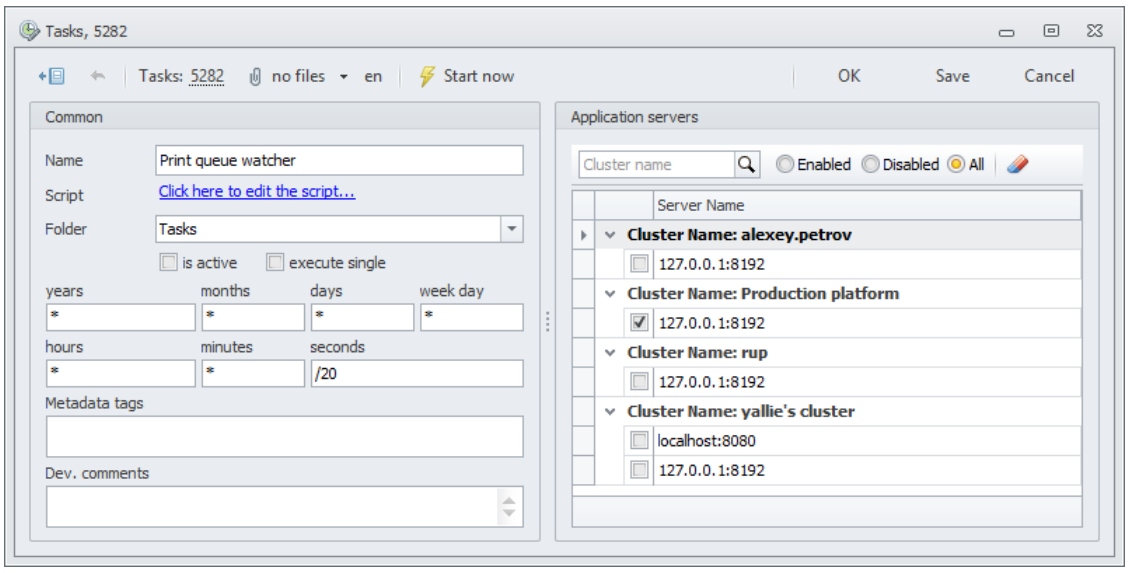
In addition to filters by *Name* of task (*Name*) and *Tags* (*Tag*), the dictionary has the filter *When did task work?*, which is activated by the flag to the right of the filter and allows finding all tasks performed at the specified time.

The dictionary list form allows to:

- run a task immediately without waiting for the scheduled start; to do this, select the *Start now* item of the context menu;
- open the script of the selected task in the edit form; to do this, select the *Edit script* item of the context menu.

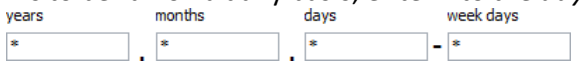
The filter *When the did the task work?* is usable if a task has changed particular data, and it is needed to find out, which task was used in doing it. All tasks are always performed on behalf of the user Task. If the history shows that the data were changed by the user Task, to find tasks, which were working in that time, it is sufficient to make use of this filter by having specified the date of alteration.

Button Start Now located in the toolbar of the task edit form allows to perform the task immediately without waiting for the scheduled start:

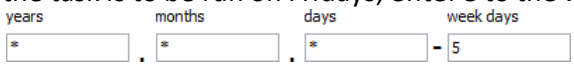


The task has the following properties:

- *Name* – name of task;
- *Script* – link to script. When creating a new task, the script is created automatically when saving the task.
  The task scripts implement the *ITaskScript* interface.
  The initial time of performing the task is transferred to the script input;
- *Folder* – a group that the task belongs to;
- *is active* – the flag indicating active tasks. If the check box is disabled, the task will not run according to schedule;
- *execute single* – the flag indicating the necessity to perform a single task instance. If it is time to start a second task instance, but it appears that the first instance is still not completed, the second instance will not be started;
- *years*, *month*, *days*, *week days*, *hours*, *minutes*, *seconds* – schedule for performing a task. Schedule fields can have the following values:
  * – any value. E. g., if the task is to be run on a daily basis, enter * to the *days* field:

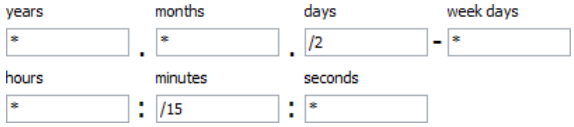


  N – positive integer. E. g., if the task is to be run on Fridays, enter 5 to the *week days* field:

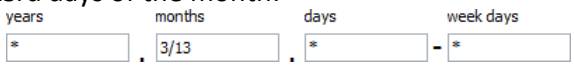


  N1, N2, N3... – a list of positive integers. E. g., if the task is to be run only in the summer, enter 6, 7, 8 to the *month* field:

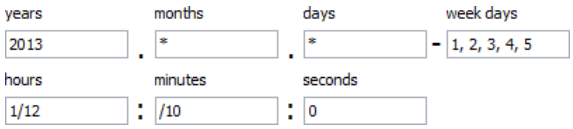
  /N – fractional value of a positive integer. The task will be performed for all possible values of date/time, if the residue of division of this values by N is zero. E. g., for the field *month*, the value /2 will relate to the even days of the month: 2, 4, 6...; for the field *minutes*, the value /15 will relate to 0, 15, 30 and 45 minutes. The task that have both these values will be performed on the even days of the month every 15 minutes:



  N1/N2 – fractional value of positive integers. The task will be performed for all possible values of date/time, if the residue of division of this values by N2 is N1. E. g., for the field *month*, the value 3/10 will relate to 3rd, 13th and 23rd days of the month:



Let's consider an example. We need the script to be executed on the application server in the year 2013 from Monday to Friday from 01:00 AM to 02:00 AM, and from the 1st to the 2nd days every 10 minutes. Then the task schedule will be as follows:



- *Metadata Tags* – tags used to describe the task functionality;
- *Dev. comments* – comments by application programmer;
- *Application servers* – list of application servers for performing the task. Servers are grouped in clusters that they belong to. Among all servers available, one must select those that the task is being planned to be performed on (if no servers selected, the task will not be performed).
  The list can be filtered by *Cluster name* according to the text entered to the "*Cluster name*" field. The list can be additionally filtered by servers using the flags:
  - *Enable* – by all servers checked;
  - *Disable* – by all servers unchecked;

- *All* – by all servers regardless of the check boxes.

To clear the filter contents and display the full list of servers, click the button .

To perform scheduled tasks in the *ConsoleServer.exe.config* configuration file of at least one application server of a cluster, enable the task planner:

```
<setting name="TaskSchedulerActive" serializeAs="String">
        <value>True</value>
</setting>
```

Totals and reports

Total drivers

Calculation of totals is performed by the total calculator in connection with totals drivers. Drivers are responsible for calculating transactions of their respective totals; the entire process is controlled by the calculator. The calculator determines a calculation interval and creates one driver instance per total in the system (the calculation involves only double-entry totals). Before the calculation is launched, each driver loads its own state at the beginning of the period. What state is—depends on a particular driver (e. g., for FIFO driver—the state is a list of the available batches).

In the process of calculation, the total calculator simultaneously loads the transactions of all totals, groups them in pairs and forms packages according to documents. The documents are processed sequentially in the order determined by the posting date. The loaded transactions are transferred to total drivers for handling; in the process, the transactions of operational totals are simply copied (since no calculation needed in this case), while the transactions of analytical totals are initially processed by the methods *CalculateIncomes*/*CalculateOutcomes* of their own drivers.

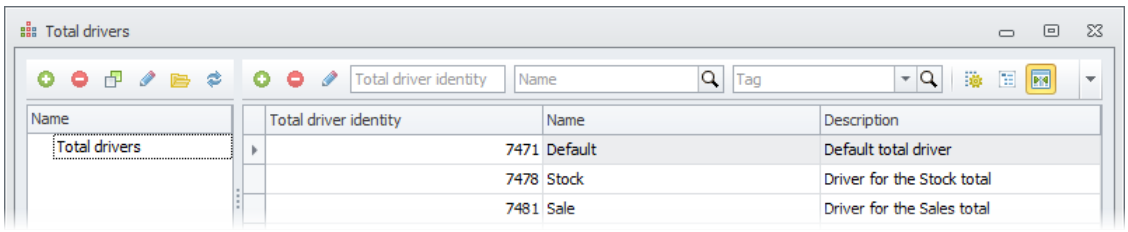
With a certain periodicity, the total calculator stores the calculated transactions in the table of round transactions and refreshes the values of the total limits calculated. In case of emergency stop of the calculation process, a restart will proceed with the calculation from the point, where the last calculation limits were set.

Each driver implements its own algorithm of management of the state total, and imposes certain requirements for the original transaction documents. If the document does not satisfy the total transaction driver requirements, results of the calculation error occurs. Errors of calculation are noncritical (not leading to divergences of the sums after calculation of results), critical (lead to divergences of the sums and the admission of wrong transactions) and fatal (causing a stop of calculation of results). Errors of the first two types are saved in a special log of errors of calculating of totals along with the numbers of documents that caused these errors. After calculating the totals, it is recommended to process the log and correct errors in documents, while re-calculating the totals will not pass without errors.

To maximize early detection of errors validation of transactions which happens at the time of saving documents is provided. It is far easier to eliminate any mistake in the document if its context isn't lost yet while author of the document remembers what was going on and can update the information. Validation during document posting is done with a special type of script —validators of transactions. These scripts check local rules for specific results (an example of such rules: in the end, the Conversion amount cannot be zero). Besides, drivers of results are engaged in validation of conductings They check the rules relevant to the outcome groups (an example of such universal rules: the transaction amount may not have fractional cents). During document posting validators of drivers outcome are applied in the first place, and then validator-scripts of all outcomes, which makes the document postings.

Any driver can be assigned to several totals simultaneously. The goal of an application programmer is to choose for the total one of the standard configuration drivers or implement his own on the basis of base classes. In operational totals, which data do not require calculation, the *Default* driver is used (ID *7471*).

The list of all total drivers can be found in the "Total drivers" dictionary:

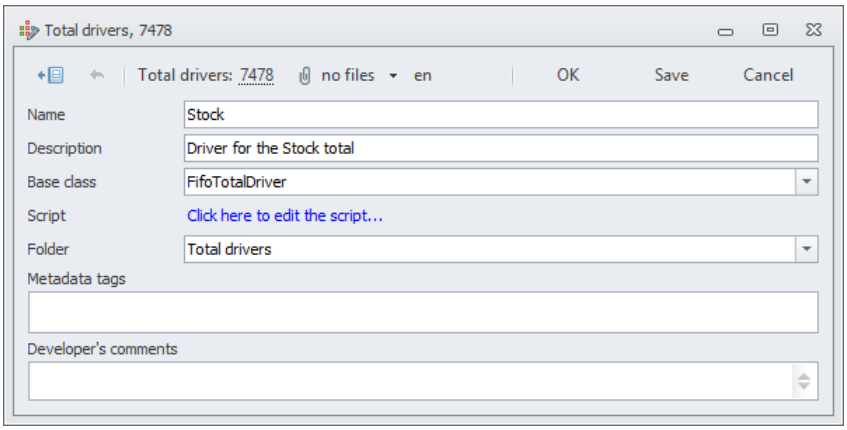


The dictionary window is divided into two parts: to the left, total drivers groups in a tree-like structure are displayed; to the right, a list of drivers from the group selected on the left.

Dictionary records can be filtered by *Name* and *Tags*.

To open a script of a selected total driver in the edit form, select the item *Edit script* in the context menu of the dictionary list form.

A total driver has the following properties:



- *Name* — name of driver;
- *Description* — description of driver;
- *Base class*— base class of driver. A base class is selected according to the specific character of the total, which analytical data are calculated by the driver. There are the following base classes:
  - *DefaultTotalDriver*— used by default for totals, which data do not require calculation;
  - *FifoTotalDriver* — implements the calculation of cost under the FIFO method (first-in, first-out). Accordingly, this is used for those total drivers that treat the cost as an analytical variable;
  - *MoneyTotalDriver* — a variant of the *FifoTotalDriver* class, which variable *Quantity* is replaced with the variable *CurrencyAmount*. This is used for those total drivers that treat money as an analytical variable;
  - *CashlessMoneyTotalDriver* — a variant of *MoneyTotalDriver*, that may be in the red. This is used for those total drivers that treat cashless funds as an analytical variable;
  - *MarginTotalDriver — a base class for the drivers of totals of sale (of goods and services) and conversion (of currencies);*
  - *CurrencyExchangeTotalDriver*— a base class for the currency exchange driver.
- *Script* — link to the script. When creating a new total driver, the script is created automatically after the driver has been saved. Click the link *Click here to edit the script...* while creating a new total driver will save the driver and reload it; after that, a script edit form will open;
- *Folder* — a group that the driver belongs to;
- *Metadata tags* — tags used to describe the driver functionality;
- *Developer's comments* — comments by the application programmer.

Scripts of total drivers are inherited from the base class specified in the property *Base class.* All base classes of totals drivers realize the *ITotalDriver* interface (from the namespace *Ultima.Totals.Calculation*).

The interface *ITotalDriver* contains the following methods and properties:

- *TotalID* of the type *long* — returns the total id;
- *LimitDateTime* of the type *DateTime* — returns the date on which the current total is calculated;
- *LimitDocumentID* of the type *LimitDocumentID* — returns the document id on which the current total is calculated;
- *DocumentBalance* of the type *decimal* — returns the balance value of the current document (the sum of all values of the variable *Sum* of its transactions);
- *BeginCalculation(ITotalCalculator transactionProcessor)* — is caused at the beginning of process of transactions calculation;
  - *transactionProcessor* — a copy of the transaction handler that is used to calculate the error messages, etc.;
- *EndCalculation()* — is caused at the end of process of the transactions calculation;
- *BeginDocument(DateTime transactionDate, long docId)* — is caused at the beginning of calculation of each document;
  - *transactionDate* — transaction date;
  - *docId* — document id;
- *EndDocument(DateTime transactionDate, long docId)* — is caused at the end of calculation of each document;
  - *transactionDate* — transaction date;
  - *docId* — document id;
- *BeginTransaction()* — is caused in the beginning of calculation of each transaction;
- *EndTransaction()* — is caused in the end of calculation of each transaction;
- *LoadTotalState(ITransactionLoader loader, DateTime limitDate, long limitDocId)* — loads the total status for the specified date (or transaction date of the specified document:
  - *loader* — transaction loader;
  - *limitDate* — transaction date, to which a total status will be loaded;
  - *limitDocId* — the document id, for which transaction date a total status will be loaded;
- *AddCompleteTransaction(DetailedTransactionValue transaction)* — adds transaction to the list of full total transactions:
  - transaction — transaction for adding;
- *CanCalculateOutcome(TransactionValue outcome)* — reports whether the current driver can independently calculate the specified account transaction to full one:
  - *outcome* — outcome transaction;
- *CalculateOutcomes(TransactionValue outcome, IEnumerable<TransactionValue> incomes)* — counts account transactions to full ones:
  - *outcome* — outcome transaction for counting;
  - *incomes* — counted income transaction;
- *CalculateIncomes(TransactionValue income, IEnumerable<TransactionValue> outcomes)* — counts account income transactions to full ones:
  - *income* — income transaction for counting;
  - *outcomes* — counted outcome transaction;
- *DetailedTransactions* типа *ICollection<DetailedTransactionValue>* — returns the calculated detailed transaction which is ready to be recorded into the database. In applied drivers of the totals this collection is available only for reading.

The DefaultTotalDriver driver serves as a base class for all other classes of drivers of totals. It realizes the *ITotalDriver* interface and provides a number of virtual methods which are redefined in descendant-classes.

The basic driver does not keep a state and does not provide any algorithm of calculation for analytical variables and dimensions, so by itself it is usually used only for operational and unbalanced totals, where there is nothing to count.

Though the driver does not count analytical data independently, it nevertheless is able to transfer the information between the pair transactions, which are filled asymmetrically. For example, if in the processed couple of transactions if there is a dimension IncomeDocumentID, which is filled only in one of two transactions, the driver will automatically copy this value into the second transaction. When copying variables in pair transaction the driver automatically changes the sign.

Other useful property of the basic driver is an ability to stratify transactions symmetrical. If when processing of pair of transactions in one of totals the stratification took place, and in another there was one transaction, the basic driver will break this transaction into the same quantity of fragments, will symmetrically break values of variables and will check their total amount remains the same.

The following properties and methods of the base class *DefaultTotalDriver* can be redefined (selectively) in a scrip:

- *LoadTotalState* — to load total state for the specified date;
- *CalculateIncomes* — to calculate income transactions to full ones;
- *CalculateOutcomes* — to calculate outcome transactions to full ones;
- *CanCalculateOutcome* — to report whether the driver can independently calculate the specified account transaction to full one;
- *BeginDocument*/*EndDocument* — is caused in the beginning/end of calculation of each document;
- *BeginTransaction*/*EndTransaction* — is caused in the beginning/end of calculation of each transaction;
- *CheckVariableValues* — the flag defines whether control of variables total amount is needed after stratification of transactions.

The collection *DetailedTransactions* is not available in the basic driver. If the driver needs to see own calculated postings which is not saved in the database yet, it can use the method *FindDetailedTransactions*.

FifoTotalDriver — a base class of the driver for the total, organized as FIFO. It keeps the state, introduces another party number (LotNo) in addition to dimensions of the total. It can count prime cost of parties according to FIFO algorithm. At an entry of the party of registration data the driver of a total fixes its prime cost and uses it at the moment of write-off, at the same time the parties are written off in the same order in what they have been credited. At write-off of quantity in minus (an expense of party, which has not been earlier credited) gives an error message of write-off and zero prime cost.

MoneyTotalDriver — a kind of the FIFO driver focused on the accounting of cash. For the accounting of quantity instead of the Quantity variable CurrencyAmount is used, there are no other differences from the basic driver.

CashlessMoneyTotalDriver — a kind of the FIFO-driver for the accounting of non-cash money. For non-cash money there is an operation of an overdraft (account balance comes to minus), which this driver does not consider as a write-off error. Instead of it the operation forms negative party with its own prime cost. At repayment of an overdraft the driver always uses prime cost of this party, in order to the zero balance in accuracy has coincided with zero balance on the amount in the account currency by quantity (currency amount). For coordination of amounts in conductings at the same time the auxiliary driver of *Conversion*, based on *MarginTotalDriver* is always used

The following methods of the base class *FifoTotalDriver* can be redefined:

- *CalculatePartialAmount* — to calculate the amount for a partial expense of party (here it is possible to set, rules of rounding, for example).

The following methods of the base class *CashlessMoneyDriver* can be redefined:

- *CalculateTransactionAmount* — to calculate the unknown amount of transaction (for example, at a current rate of the Central Bank).

For calculation of sales of goods and currencies, the totals *Sale* and *Conversion* are used, which drivers are based (usually, without any changes in logic) on the driver *MarginTotalDriver*. This driver always handles two pairs of transactions: receipt and expenditure. The pairs always have a common quantity variable, but, as a rule, their amounts differ.

The *Sale* total acts as follows: receipt of an article at cost followed by sale of the article at selling price. The difference between the expenditure and the receipt is proceedings from the sale of the article. This is how sale transactions look like:

	Total: Stock	**Total: Sale**	**Total: Agents balances**
1. Receipts (no amounts specified)	− Quantity, No amount	+ Quantity, No amount	
2. Expenditure (together with the amounts)		− Quantity, − Amount	+ Amount

For the receipt pair of transactions, the *Stock* total driver (working on the base of *FifoTotalDriver*) calculates the receipts, that is the cost, following the FIFO algorithm. The *Sale* driver copies this amount into the pair transactions. In the expenditure pair of transactions, the amount is already specified: this is the selling price of the article. The expenditure transactions write off the article from *Sale* and record the debt to *Agent balance*; the *Sale* will contain the difference between the receipts and expenditure.

The *Conversion* total is used for harmonization of amounts when transferring the money between accounts and buying/selling foreign currencies. The total handles two pairs of transaction at once; the result is similar: the difference between receipts and expenditure accumulates in the total. Below is a scheme for *Conversion* transactions to transfer the money between accounts:

	Total: Bank accounts	**Total: Conversion**	**Total: Bank accounts**
1. Receipt (together with the amounts)	− Quantity, − Amount	+ Quantity, + Amount	
2. Expenditure (no amounts specified)		− Quantity, No amount	+ Quantity, No amount

The receipt transaction shows the amount (a money equivalent of the amount in the given currency); the expenditure transaction shows no amounts. The driver of the source account either provides the cost of the currency batch or uses a money equivalent (if a debit of the account into the red occurs, and a new negative batch having its own cost is created). In the second pair of transactions, the cost can be provided either by the recipient account (if it is needed to close the negative batch under this account) or uses the same cost that appears in the receipt pair of transactions. Most commonly, when both bank accounts have a positive balance, the transfer of the money between the accounts does not affect the cost of the currency: the cost is transferred via the Conversion total without any changes. But if there is an overdraft on either side, the *Conversion* total acts as a buffer that harmonizes costs of batches.

Another scheme for *Conversion* transactions can be used when buying/selling currency from the agent's balance:

	Total: Bank accounts	Total: Conversion	Total: Agents balances
1. Receipt (together with the amounts)	− Quantity, − Amount 1	+ Quantity, + Amount 1	
2. Expenditure (together with the amounts)		− Quantity, − Amount 2	+ Amount 2

Amounts shown in the receipt pair of transactions, as in the previous case, are used only if the account is in the red. If the balance of the source account is positive, the driver of the source account will replace the Amount 1 with the real cost of the money batch. As in the previous case, the *Conversion* total will accumulate the difference between the expenditure and the receipt, which can be considered as proceeds (or losses) from currency exchange.

This driver is used only for operations of currency exchange with preservation of prime cost of the currency. It is similar to the simplified driver *MarginTotalDriver*: it also works with two pairs of postings at a time, but it does not require write-off of quantity to zero and completely ignores unmatched dimension. It allows crediting dollars on a total, and to write off euro from it. In a simple case the postings scheme on a total *Currency exchange* looks like this:

	Total: Bank account ($)	Total: Currency exchange	Total: Bank account (€)
1. Income (with sums)	− Quantity$, − Sum	+ Quantity$, + Sum	
2. Expense (without sums)		− Quantity€, Without sum	+ Quantity€, Without sum

Here the Quantity$ — quantity of dollars, Quantity€ — respectively, euro (numbers, of course, do not coincide). In the first posting the sum is specified (prime cost in account currency) which is necessary only when the account-source become negative. Under normal circumstances, if the balance of the account-source is positive driver of the *Bank account* reports the prime cost of this currency party. If the money which is written off from the account-source belongs to several different parties with different cost — the posting is stratified, and similar to it the receipt posting is stratified on a total *Currency exchange*. If the posting on the receiving account is stratified, then the pair posting to it *Currency exchange* is also stratified in the same way.

On the receiving account there can be a negative balance as well, because of what the prime cost of the currency in the second pair of postings partially or will be completely taken from the receiving account (to close negative party in zero). In this case the total *Currency exchange* in the second pair of postings uses this prime cost, and the difference will settle on a total *Currency exchange*.

Most often it is more convenient to consider this difference on the *Conversion* total — in the same place where profits and losses from all other operations with currency. It is easy to achieve it if the second pair of postings will be to driven through the *Conversion*:

	Total: Bank account ($)	Total: Currency exchange	Total: Conversion	Total: Bank account (€)
1. Income (with sums)	− Quantity$, − Sum	+ Quantity$, + Sum		
2. Expense (without sums)		− Quantity€, Without sum	+ Quantity€, Without sum	
3. Conversion (without sums)			− Quantity€, Without sum	+ Quantity€, Without sum

The difference of prime cost of parties of the currencies will settle on *Conversion*, the delta will remain zero on a total *Currency exchange*.

Consider an example situation, when it is needed to sell from a store a production of a particular product consisting of details; the stock is calculated in the *Stock* total. To do this, you will need to create in the system an intermediate total called *Production*; the details will be written off from the *Stock* to this total; the product assembled from the details will be recorded to the *Stock* from this total as well:

- *Stock*-> *Production* // a detail;
- *Stock* -> *Production* // one more detail;
- *Stock* -> *Production* // one more detail;
- *Production*-> *Stock* // assembled product;

It is needed to code a total driver that will calculate the cost of the product assembled from the details. *DefaultTotalDriver* will be a base class, from which the total driver will be inherited.

It is necessary to avoid database accesses in the total driver.

When coding the driver, it is needed to work out the following problems:

1. Transactions must be done by the document that allows determining which details were used for the production. To avoid accessing the database, a queue is formed: Making an empty queue for a queued document can be done by redefining the virtual methods of the driver *BeginDocument*/*EndDocument*.
2. Further, calculate the details that came from the store by using the method *CalculateOutcomes*:

```
public override ICollection<DetailedTransactionValue> CalculateOutcomes(
    TransactionValue outcome, IEnumerable<TransactionValue> incomes)
```

   Method's input parameters:
   - *outcome* – write-off transaction. This must be calculated as round and, if necessary, split into several transactions;
   - *incomes* – receipt transaction. Probably, already split by the pair driver and calculated as round.

   Since the production doesn't possess any split logic, the split must be done by the basic driver upon the model of pair transactions. But first we must calculate the cost. If to call the *Default* basic driver method right away, the method will fail to obtain *Amount*, and an exception will be shown:

```
outcome.Amount = CalculateAmount(); // calculate the cost
return base.CalculateOutcomes(outcome, incomes); // perform split
```

3. Finally, as done above, we need to calculate the cost of the assembled product via the *CalculateIncomes* method:

```
public override ICollection<DetailedTransactionValue> CalculateIncomes(TransactionValue
income, IEnumerable<TransactionValue> outcomes)
```

As a result, we obtain the driver:

```
decimal docIncomeAmount = 0;

public override void BeginDocument(DateTime transactionDate, long docId)
{
    docIncomeAmount = 0;
    base.BeginDocument(transactionDate, docId);
}

public override ICollection<DetailedTransactionValue> CalculateOutcomes(
    TransactionValue outcome, IEnumerable<TransactionValue> incomes)
{
    var result = base.CalculateOutcomes(outcome, incomes);
```

```
    var stock = outcome as StockTransaction;

    if (stock != null && stock.Quantity < 0 && incomes.Count() > 0 &&
        (incomes.First() is ProductionTransaction))
    {
        docIncomeAmount -= result.Sum(x => (x as StockDetailedTransaction).Amount ?? 0);
    }

    return result;
}

public override ICollection<DetailedTransactionValue> CalculateIncomes(
    TransactionValue income, IEnumerable<TransactionValue> outcomes)
{
    var result = base.CalculateIncomes(income, outcomes);

    if (result.Count > 1)
    {
        AddError(result.First(), @"Single income Prod->Stock expected: {0}...{1}",
            result.First(), result.Last());
    }

    var stock = income as StockTransaction;

    if (stock != null && stock.Quantity > 0 && outcomes.Count() > 0 &&
        (outcomes.First() is ProductionDetailedTransaction))
    {
        (result.First() as StockDetailedTransaction).Amount = docIncomeAmount;
    }

    return result;
}
```

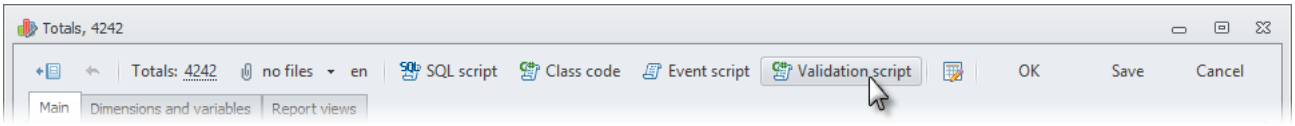
Transaction validator for the driver (for a single product in a document):

```
public void ValidateTransactions(TransactionPairCollection transactionPairs,
    TransactionCollection transactions)
{
    long ProductID = -1;
    foreach (var pair in transactionPairs)
    {
        var outcome = pair.OutcomeValue as StockTransaction;
        if (outcome != null)
        {
            if (!outcome.IsValueNull("Amount"))
            {
                throw new UltimaException(@"Amount variable of an outcome stock
                    transaction mustn't be set.");
            }
        }

        var income = pair.IncomeValue as StockTransaction;
        if (income != null && pair.OutcomeValue.IsComplete)
        {
            if (!income.IsComplete)
            {
                throw new UltimaException(@"Income stock transaction must be
                    complete when outcome transaction is complete.");
            }
        }
```

```
        if(pair.OutcomeValue is ProductionTransaction)
        {
            if(ProductID == -1)
            {
                ProductID = (pair.OutcomeValue as ProductionTransaction).ProductID;
            }

            if((pair.OutcomeValue as ProductionTransaction).ProductID != ProductID)
            {
                throw new UltimaException(@"In document must be only one product");
            }
        }

        if(pair.IncomeValue is ProductionTransaction)
        {
            if(ProductID == -1)
            {
                ProductID = (pair.IncomeValue as ProductionTransaction).ProductID;
            }

            if((pair.IncomeValue  as ProductionTransaction).ProductID != ProductID)
            {
                throw new UltimaException(@"In document must be only one product");
            }
        }
    }
}
```

Transaction validators

Transactions validators – the scripts used to check the transactions for validity (correctness). Validators are closely connected with the used totals drivers. The list of all validators can be found in the dictionary “Scripts”. Besides, the validator of a particular total can be opened from its edit form:



Validator script is carried out when saving the document after transaction scripts and allows checking correctness of transactions before their saving, giving an exception in case of an error. This, in turn, allows reducing errors quantity when calculating of the detailed transactions by the total driver.

For example, by means of validators it is possible to check the following restrictions, imposed on total transaction by the used drivers:

- it is forbidden to do the full account movement on the total, on which the driver *FifoTotalDriver* is used;
- it is forbidden to do an incomplete income movement or to do the movement bypassing the total conversion on a total, in which the *CashlessMoneyTotalDriver* driver works.

Total validation scripts realize the *ITransactionValidator* (from the namespace *Ultima.Totals*), which contains the only one method:

- *ValidateTransactions(TransactionPairCollection transactionPairs, TransactionCollection transactions)* – carries out check of transactions, it can be redefined by the applied developer:
  - *transactionPairs* – collection of pairs of transactions for checking;
  - *transactions* – collection of transactions for checking.

Totals drivers also realize the *ITransactionValidator* interface, that allows coding in them the universal rules, suitable for the whole groups of totals. In order not to repeat the same rule in a script-validator on each total, it can be transferred directly to the driver. For this purpose in a driver script it is necessary to redefine the virtual method *ValidateTransactions*, having exactly the same signature as it is described above.

Example of the universal rule existing for all totals: the sum of transaction can not have a fractional part, less than one kopeck. This rule is checked by the DefaultTotalDriver driver. If necessary this rule can be toughened (for example, to forbid kopecks in the sums at all) or to be abolished completely. To simplify the work with this rule, the DefaultTotalDriver driver defines the virtual method *ValidateAmount(TransactionValue transaction)*. To cancel the rules about fractional kopecks it is enough to redefine an empty method *ValidateAmount* in the driver script and not to transfer the control to the base class method. Like this:

```
protected override void ValidateAmount(TransactionValue transaction)
{
    // do nothing: any amount is valid
}
```

Column providers

 The Column providers are used to create additional levels of report details (data grouping).

There are initially two Column providers in the system by default: documents and dictionaries.

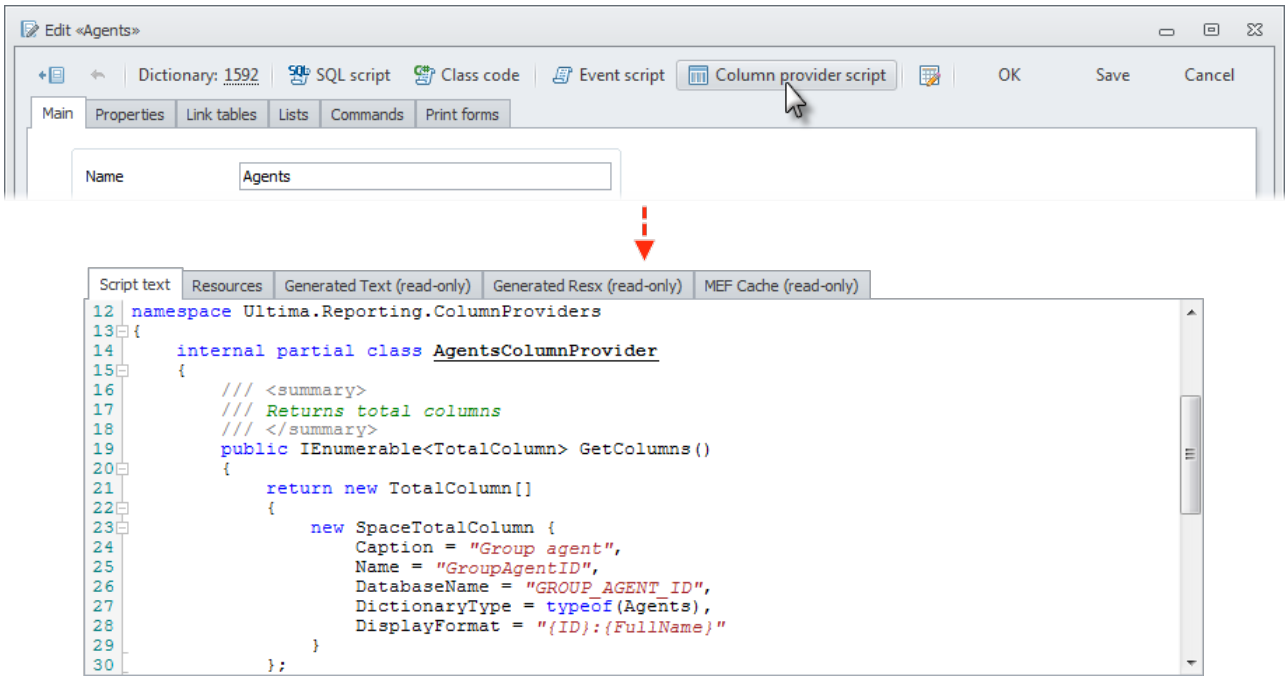
The provider of documents provides a possibility to detail the reports by:

- time (periods):
  - days;
  - weeks;
  - months;
  - quarters;
  - years;
- directly by the documents.

The provider of dictionaries provides a possibility to detail the reports in addition to the values of the dictionary itself by the values of its properties-links.

If this level of details is insufficient for some dictionary being furnished by the column provider by default, an own column provider can be created for it. Click *Column provider script* in the dictionary edit form to create the column provider. During creation of a new dictionary, the provider of its columns is not created by default.

The list of all column providers can be found in the dictionary "Scripts". Besides, the column provider of particular dictionary can be opened from its edit form:



The scripts of column providers implement *ITotalColumnProvider* interface, (from *Ultima.Server.Reporting.ColumnProviders namespace*) which implements in turn the following methods:

- *GetColumns* returns enumeration of the columns, which will be interpreted by the report mechanism as additional dimensions:

```
public IEnumerable<TotalColumn> GetColumns()
{
	return new TotalColumn[]
	{
		// dimension
		new SpaceTotalColumn {
			Name = "GroupAgentID",
			Caption = "Group agent",
			DatabaseName = "GROUP_AGENT_ID",
			DictionaryType = typeof(Agents),
			DisplayFormat = "{ID}:{FullName}"
		}
	};
}
```

SpaceTotalColumn class describes the columns of the total dimensions, is derived from *TotalColumn class,* has the following properties:

- *Name*, type of *string* returns or sets the column name;
- *Caption*, type of *string* returns or sets column caption;
- *DatabaseName*, type of *string* returns or sets the column name in the database;
- *ColumnType*, type of *TotalColumnTypes* returns or sets the column type, can assume the values:
  - *Identity* – ID;
  - *Number* – number;
  - *Date* – date;
- *Parent*, type of *TotalColumn* returns or sets the column parent;
- *RootColumn*, type of *TotalColumn* returns the root parent of the column;
- *TreeLevel*, type of *long* returns or sets the level of column parent;
- *TransactionsOnly*, of *bool* type returns or sets the flag indicating that the column values are analytical (computing);

- *FullName*, type of *string* returns or sets a full name of the column;
- *FullCaption*, type of *string* returns or sets a full caption of the column;
- *FullDatabaseName*, type of *string* returns or sets a full name of the column in the database;
- *Nullable*, type of *bool* returns or sets the flag indicating if the column values can be *Null*.
- *DictionaryType*, type of *Type* returns or sets the dictionary being a dimension of the column;
- *DisplayFormat*, type of *string* returns or sets a format to display column values;

- *GetJoinExpression(TotalColumn column)* returns JoinExpression for the *column*:

```
public JoinExpression GetJoinExpression(TotalColumn column)
{
	return new JoinExpression {
		TableName = "ARTICLE_GROUPS",
		ColumnName = "ID" };
}
```

JoinExpression class describes JOIN expression, has the following properties:

- *TableName*, type of *string* returns or sets the table name in the database;
- *ColumnName*, type of *string* returns or sets the table column name;
- *Type*, type of *string* returns or sets the type of (operator kind) JOIN expression, can assume the values:
  - LEFT;
  - RIGHT;
  - INNER;
  - OUTER.

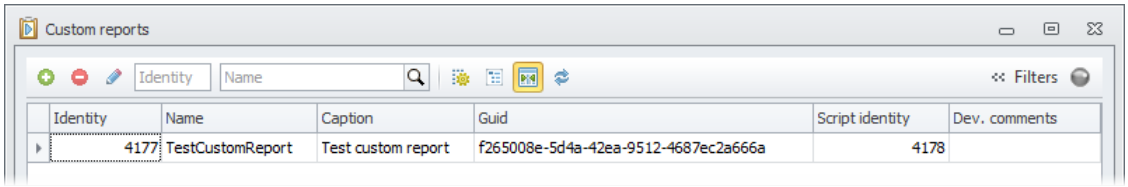
In the example above, JOIN expression will have the form:

```
// select * from ARTICLES A
JOIN ARTICLE_GROUPS AG on AG.ID = A.GROUP_ID
```

Custom reports

Need for user reports arises when the totals reports are lacking functionality, and it is necessary to implement a non-standard report, e. g., with additional data specification, which is unreachable when dealing with dimensions.

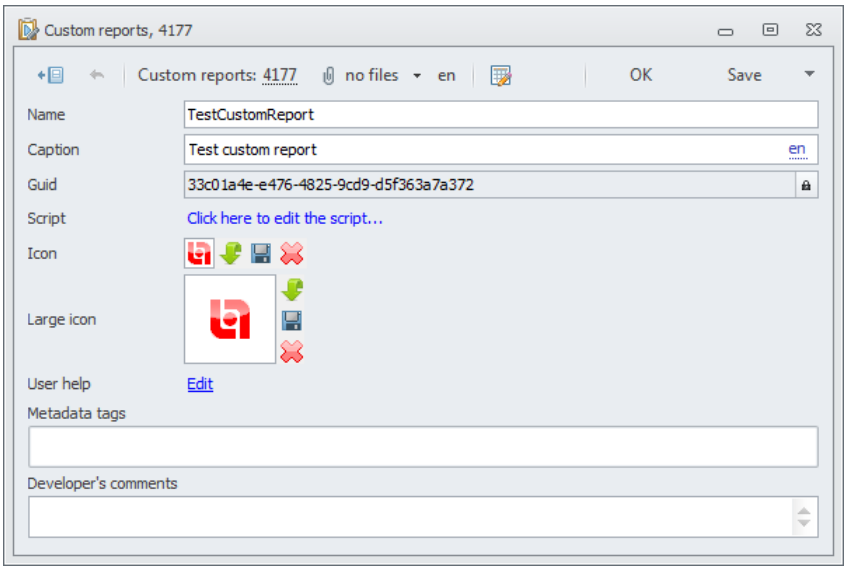
The list of all user reports can be found in the dictionary "Custom reports":



he dictionary records can be filtered by report *Name* (*Name*).

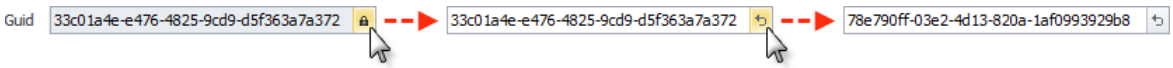
The script of the user report selected in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.

Button in the toolbar of the user report edit form allows opening of the report (a form of report parameters). The functionality is available only for already created reports:



The user report has the following properties:

- *Name* – report name;
- *Caption* – the report name displayed in the screen forms;
- *Guid* – used to identify a menu item.
  Guid is generated automatically at random and, if necessary, (in case of coincidence with Guid of another object) can be changed:



- *Script* – link to the script. In case of creation of new user report, the script is created automatically upon its saving. Click on the link *Click here to edit the script...* during creation of new user report will result into saving of the report and its reloading, after that the script edit form will open;
- *Icon* – report icon (with the size of 16 x 16 pixels).
  The buttons to the right of icon preview area allow:
  – loading the icon;
  – saving the icon previously downloaded to the computer;
  – deleting the icon;
- *Large icon* – a large icon (with the size of 32 x 32 pixels);
- *User help* – a comment to the report, which end the user can see as a hint, dropping down in case of mouseover at the report. The comments are entered for each of system languages in the form opened by clicking the link;
- *Metadata tags* – tags used to describe report functionality;
- *Developer's comments* – comments of application developer.

The user reports employ the functionality of totals reports, implemented with the system. This implies the task of the application developer during writing of the user report - to provide the data of own report similarly to the total data, which can be handled by the report mechanism.

The scripts of user reports implement *IReportDataSource* (from *Ultima.Server.Reporting.DataSources* spacename), which, in its turn, implements the following methods:

- *GetTransactionsSql* returns a query for transactions (transactions);
- *GetBalanceSql* returns a query for balances (remains);
- *GetReportColumns* returns the report columns (dimensions and variables):

```
public IEnumerable<TotalColumn> GetReportColumns()
{
```

```
        return new TotalColumn[]
        {
                // date
                new DateTotalColumn {
                        Name = "TransactionDate",
                        Caption = "Process date",
                        DatabaseName = "TRANSACTION_DATE",
                        TransactionsOnly = true
                },
                // document
                new DocumentTotalColumn {
                        Name = "Document",
                        Caption = "Document",
                        DatabaseName = "DOCUMENT_ID",
                        TransactionsOnly = true
                },
                // dimension
                new SpaceTotalColumn {
                        Name = "ProductID",
                        Caption = "Product",
                        DatabaseName = "PRODUCT_ID",
                        DictionaryType = typeof(Goods),
                        DisplayFormat = "{ID}:{Name}"
                },
                // variable
                new VariableTotalColumn {
                        Name = "Amount",
                        Caption = "Amount",
                        DatabaseName = "AMOUNT",
                        Modifiers = new[] {
                                VariableTotalColumnModifier.In,
                                VariableTotalColumnModifier.Add,
                                VariableTotalColumnModifier.Sub,
                                VariableTotalColumnModifier.Out
                        }.ToList()
                }
        };
}
```

The base class of the report column *TotalColumn* has the following properties:

- *Name*, type of *string* returns or sets the column name;
- *Caption* of *string* type, returns or sets column caption;
- *DatabaseName* of *string* type returns or sets the column name in the database;
- *ColumnType* of *TotalColumnTypes* type returns or sets the column type, can assume the values:
  - *Identity* – ID;
  - *Number* – number;
  - *Date* – date;
- *Parent* of *TotalColumn* type returns or sets the column parent;
- *RootColumn* of *TotalColumn* type returns the root parent of the column;
- *TransactionsOnly* of *bool* type returns or sets the flag indicating that the column values are analytical (computing);
- *FullName* of *string* type returns or sets a full name of the column;
- *FullCaption* of *string* type returns or sets a full caption of the column;
- *FullDatabaseName* of *string* type returns or sets a full name of the column in the database.

The *DimensionTotalColumn* class describes the columns of dimensions, is derived from *TotalColumn,* has the following properties:

- *Nullable*, type of *bool* returns or sets the flag indicating if the column values can be *Null*.

The *DateTotalColumn* class describes the columns of temporary dimensions, is derived from *DimensionTotalColumn* class.

The *DocumentTotalColumn* class describes the columns of dimensions-documents, is derived from *DimensionTotalColumn* class.

The *SpaceTotalColumn* class describes the columns of other dimensions, is derived from *DimensionTotalColumn* class, has the following properties:
- *DictionaryType* of *Type* type returns or sets the dictionary being a dimension of the column;
- *DisplayFormat* of *string* type returns or sets a format to display column values;
- *TreeLevel* of *long* type returns or sets the level of column parent.

The *VariableTotalColumn* class describes the columns of variables, is derived from *TotalColumn* class, has the following properties:
- *Modifiers* of *VariableTotalColumnModifier* type returns or sets a list of modifiers for the variable, can assume the values:
  - *Default* = 0;
  - *In* = 1;
  - *Add* = 2;
  - *Sub* = 3;
  - *Out* = 4.

- *GetSortOrderColumns* returns a set of columns for sorting of transactions:

```
public IEnumerable<string> GetSortOrderColumns()
{
        return new[] { "TransactionDate", "Document" };
}
```

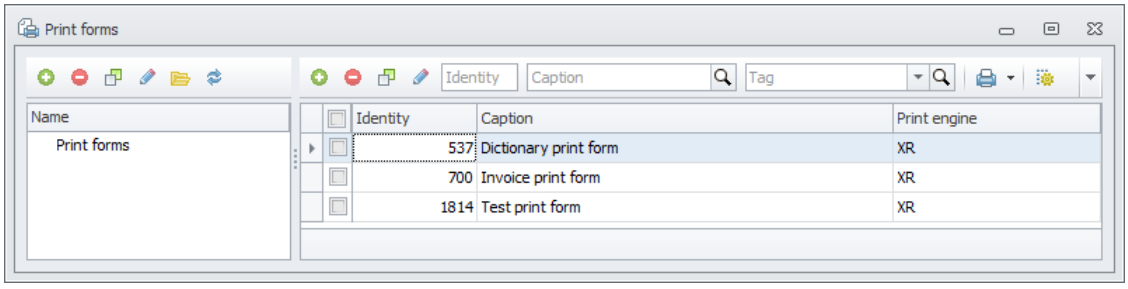
- *GetCalculatedTransactionsDate* returns a date, for which the transactions are calculated;
- *GetActualTransactionsDate* returns a date of transactions actuality (the earliest date, on which the changed document exists).

Print forms

Print forms are used to bring system objects for printing (to printer or export to the file). They were briefly described in the first chapter in the section Print forms. Print form represents the template and a script providing data to fill the template. XtraReports library-engine is engaged in processing of a template (documentation to it is available on the official website of the developer).

The printing is available both from a list form, and editing form in dictionaries and documents. Moreover, a set of forms to be printed in a list form and in an editing form can both differ and overlap:
- when printing from the edit form the only editable object is brought for printing — dictionary record or a document - opened in the edit form;
- when printing from a list form a list of objects is brought for printing (one or more), marked by flags in it.

List of all printed forms can be found in the dictionary Print forms:

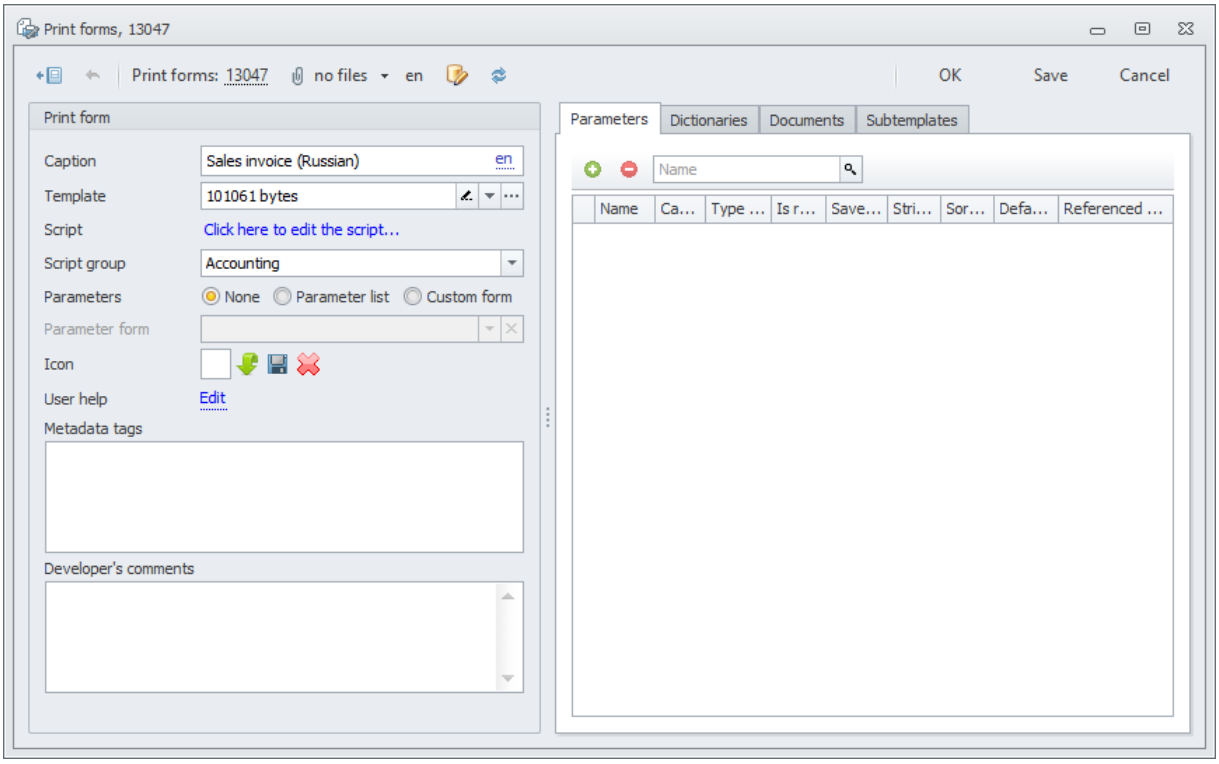


The dictionary window is divided into two parts: on the left the print forms tree of groups is displayed, on the right — the list print forms of the group chosen from the left.

Dictionary records can be filtered by the *Name of a printing form displayed in screen form* (*Caption*) and *Tags* (*Tag*).

To open a script of the chosen dictionary list command print form is possible in the form of editing directly from a list form of the dictionary, chosen the item *Edit script* in a context menu.

Print form has the following properties:



- *Caption* — name of a printing form displayed in screen form;
- *Template* — template of a print form. For each engine a personal template is used which is chosen automatically at the choice of the engine. Template control element displays its volume and allows:
  — to open the template edit form (available only for the engine *XtraReports*);
  — to save the template into the file;
  — to load the template from the file.
  Each of the engines has its own forms (methods) of editing its template, they will be described in appropriate sections;
- *Script* — link to the script. During creation of a new print form the script is created automatically at its saving. by clicking the link *Click here to edit the script...* during creation of a new print form it will lead to the saving of a form and its reset, then the form of script editing will be opened.
  Scripts of print forms are inherited from the class *PrintFormBase*, which in turn realizes the *IPrintForm* interface. More detail about them will be written further;
- *Script group* — group which possesses a print form;
- *Parameters* — the use of additional parameters before running a print form:
  - *None* — additional parameters will not be requested;
  - *Parameter list* — creation of a print form will be preceded by opening of a standard form (it is generated by the system Ultimate AEGIS® automatically), in which it will be offered to the user to fill a number of parameters;

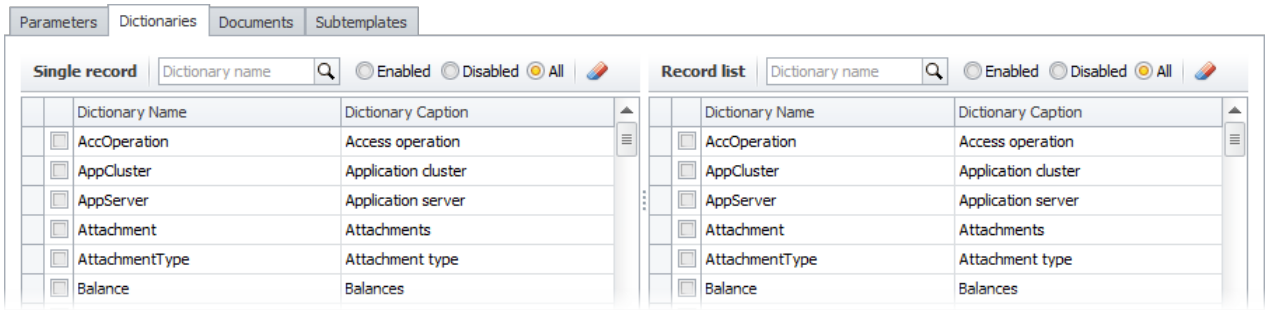
  - *Custom form* — creation of a print form will be preceded by opening of a special additional form (designed by the applied developer), in which it will be offered to the user to fill a number of parameters.
- *Parameter form* — special form with additional parameters. It is necessary to choose it in case in the item *Parameters* the option *Custom form* is chosen. This form has to be previously designed by the applied developer, for example, in Visual Studio, and is placed in the public client module (in detail the process is described in the section Request forms of parameters of interactive commands);
- *Icon* — print form icon (in size 16 x 16 pixels).
  Key buttons to the right of the preview area of an icon allow:
  — to load an icon;
  — to save the icon loaded earlier on the computer;
  — to remove an icon;
- *User help* — a comment to the command which the end user can see in the form of the hint (hint) which is dropping out at mouseover at the command. The comment is added for each language of the system which is opening by clicking the link in the form;
- *Developer's comments* — applied developer comments;
- *Metadata tags* — tags, used to describe the functionality of the print form.

A number of parameters of a print form is grouped in semantic tags.

In the tab "Parameters" additional parameters are listed, that are used in the construction/filling of the print form. All these parameters are used (on a standard form) in case in the item *Parameters* the option *Parameter list* is chosen. Parameters can be filtered by *Name* in according to the text added into the field “Name”*Name*". Each parameter has:

- *Name* — parameter name;
- *Caption* — name displaying in screen forms;
- *Type Identity* — parameter type (for more details see the section Data types);
- *Is Required* — flag, indicating whether the parameter is required to fill;
- *Save History* — flag, indicating the need to remember the last user-added value;
- *String Size* (available for data types *Text* and *String*) — limits the parameter size in specified value;
- *Sort index* — index, by which the parameters in a screen form will be sorted. As index values any integers can be used. Parameters will be ordered in the form from top to down in increasing order of the index;
- *Default Value* (is available for all data types except *Binary*) — parameter value by default which is used in the form of additional parameters;
- *Referenced Dictionary ID* (is available for data types *Long*)– dictionary ID (object), to which the parameter is a link.

In the tab “Dictionaries” a list of dictionaries is given for which this print form is used:



The list represents the list of all dictionaries, in which the flags can be marked by those, for which this print form will be used. The tab is divided into two parts:

- in the left part of the tab "Single record" the dictionaries are listed, in which a print form is used for printing of one record of the dictionary. This print form will be available for printing from an editing form of records of the chosen dictionaries;

- in the right part of the tab "Records list" the dictionaries are listed, in which a print form is used for printing of several dictionary records (marked by flags in its list form). Respectively, this print form will be available for printing of the chosen dictionaries from the list form .

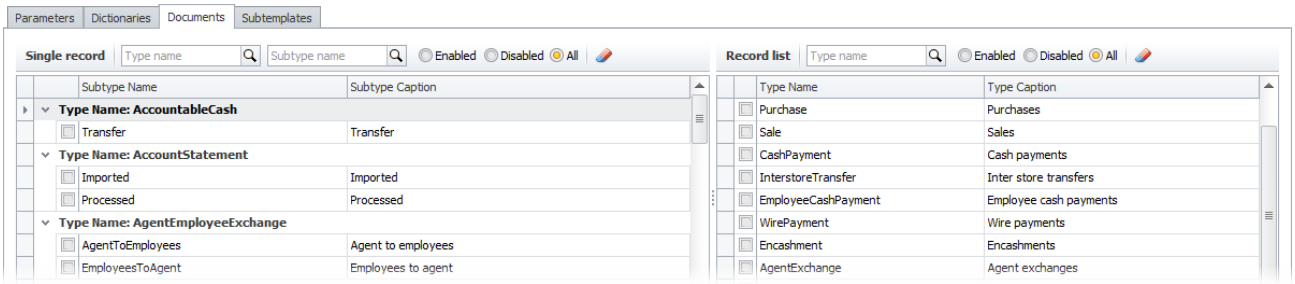
The same print form can be used both for printing of one (Single record), and for printing of several (Records list) dictionary records. In this case at an identical template in a script of a print form two different methods should be used for filling it with data.

Lists can be filtered by *Dictionary name* in according to the text added into the field *Dictionary name*. Also, the lists can be filtered in addition by flags:

- *Enable* — in all dictionaries marked by the flags;
- *Disable* — in all dictionaries not marked by the flags;
- *All* — in all dictionaries regardless of the set flag.

To clear the contents of the filter and to display the complete list of dictionaries is possible by clicking the key button .

In the tab “Documents” a list of documents (types and subtypes) is given for which this print form is used:



The list represents the list of all documents types and subtypes, in which the flags can be marked by those, for which this print form will be used. The tab is divided into two parts:

- in the left part of the tab “Single record” the *documents types and subtypes* are listed, grouped in type in which a print form is used for printing of one document. This given print form will be available for printing from an editing form of documents of the chosen subtypes;
- in the right part of the tab “Records list” the *types of documents* are listed, in which a print form is used for printing of several documents (marked by flags in its list form). Respectively, this print form will be available for printing of the chosen types from the list form of the documents .

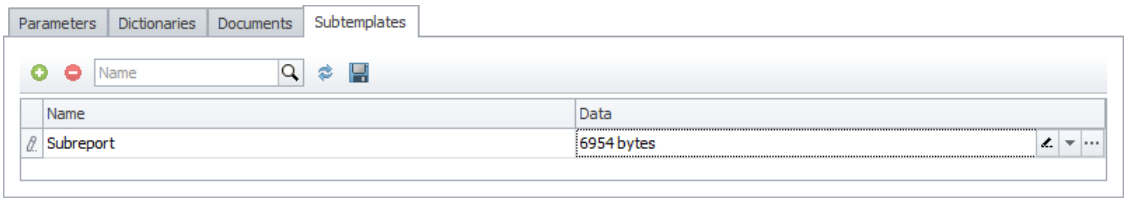
The same print form can be used both for printing of one (Single record), and for printing of several (Records list) documents. In this case at an identical template in a script of a print form two different methods should be used for filling it with data.

List “Single record” can be filtered by the *Type name* or *Document subtype name* according to text added into the fields *Type name* or *Subtype name*. List “Records list” can be filtered by *Type name of the document* according to text added into the field *Type name*. Also, the lists can be filtered in addition by flags in subtypes/types of the documents:

- *Enable* — in all subtypes/types marked by the flags;
- *Disable* — in all subtypes/types not marked by the flags;
- *All* — in all subtypes/types regardless of the set flag.

To clear the contents of the filters and to display the complete lists of subtypes/types of the documents is possible by clicking the key button .

In the tab "Subtemplates" is a list of auxiliary templates (subtemplates) of a print form is given, used in the main template by the engine *XtraReports*:



Subtemplates can be filtered by the *Name* according to the text added into the field “Name”*Name*.

Subtemplates can be added or deleted by the corresponding key buttons in the toolbar of the tab. To save added, modified or removed subtemplate it is necessary to click the key button .

Each subtemplate has:

- *Name* — name;
- *Data* — actually the subtemplate, displayed in the control element with the same functionality , as well as the template *Template*.

Scripts of print forms are inherited from the class *PrintFormBase* (from the namespace *Ultima.Server.Printing*), which in turn realizes the *IPrintForm* interface (from the namespace *Ultima.Printing*).

The *IPrintForm* interface realizes the following methods:

- *SlimTable GetDataTemplate(string subReportName = null)* — returns the data template of a print form:
  - *subReportName* — subtemplate name (optional parameter);
- *SlimTable GetData(Type dictionaryType, long id, IDictionary<string, object> parameters)* — returns data of a print form for the specified dictionary record:
  - *dictionaryType* — dictionary type;
  - *id* — dictionary record id;
  - *parameters* — additional parameters;
- *SlimTable GetData(IDictionaryRecord record, IDictionary<string, object> parameters)* — returns data of a print form for the specified dictionary record:
  - *record* — dictionary record;
  - *parameters* — additional parameters;
- *SlimTable GetData(Type dictionaryType, long[] ids, IDictionary<string, object> parameters)* — returns data of a print form for the specified dictionary records:
  - *dictionaryType* — dictionary type;
  - *ids* — dictionary records ids;
  - *parameters* — additional parameters;
- *SlimTable GetData(IDictionaryTable records, IDictionary<string, object> parameters)* — returns data of a print form for the specified array of the dictionary records:
  - *record* — dictionary records;
  - *parameters* — additional parameters;
- *SlimTable GetData(long id, IDictionary<string, object> parameters)* — returns data of a print form for one document:
  - *id* — document id;
  - *parameters* — additional parameters;
- *SlimTable GetData(long[] ids, IDictionary<string, object> parameters)* — returns data of a print form for several documents:
  - *id* — documents ids;
  - *parameters* — additional parameters;

- *SlimTable GetData(IDictionary<string, object> parameters)* — returns data of a print form (does not receive any object at an input):
  - *parameters* — additional parameters.

The class *SlimTable* — a container of data for printing forms — realizes the *IEnumerable* and *ITypedList* interfaces. Its property *TableName* — a container name. In case if a print form has subtemplates (*subtemplate*) names of containers have to coincide with subtemplates names.

Thus, in a print form script it is necessary to set the data template of the print form and to fill it with data, having redefined methods *GetDataTemplate* and *GetData*:

```
[Import]
private IDocumentManager DocumentManager { get; set; }

public override SlimTable GetDataTemplate(string subReportName = null)
{
	var template = new SlimTable();
	template.Fields["BuhNo"] = typeof(string);
	template.Fields["BuhDate"] = typeof(DateTime);
	template.Fields["CurrentDate"] = typeof(DateTime);

	return template;
}

public override SlimTable GetData(long recordId, IDictionary<string, object> parameters)
{
	var doc = DocumentManager.GetDocument(recordId) as PurchaseDocument;

	var buhNo = doc.AccountingNo;
	var buhDate = doc.AccountingDate;
	var currentDate = DateTime.Now;

	var data = new SlimTable("Report");
	data.Add(new
	{
		BuhNo = buhNo,
		BuhDate = buhDate,
		CurrentDate = currentDate,
	});

	return data;
}
```

To specify the heading of a print form (which is visible in a preview window), it is necessary to specify the SlimTable.Caption property. If the heading is not specified, by default the name of a print form will be used in the user language with the number of the document (if any).

Integration tests

Integration tests are scenarios of check of system work, which have to lead to the predetermined results. Unlike the modular tests, working with most isolated program fragments, integration tests usually cover several interacting processes of different layers of the system. Large functional blocks are the purpose of the verification of integration tests, such as commands over the documents, fragments of business processes, etc. The set of all integration tests represents the specification describing correctly working system. In ideal situation all functionality of the system has to be covered with such tests.

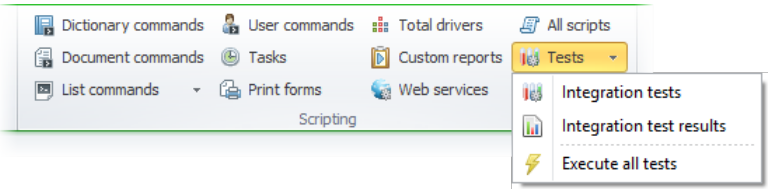
Carrying out the integration tests, the system saves the report on results of the performance: list of tests; time of their performance; mistakes that occurred at their start. Test execution without errors means that the system operates in accordance with its specifications.

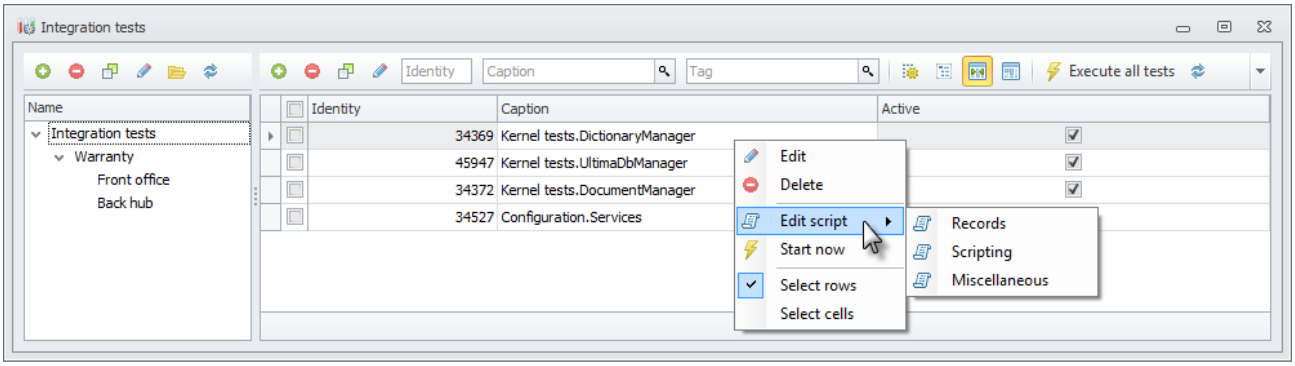
Realization of a new business process, for which the detailed Test Task is developed, has to be accompanied by writing of the integration tests covering all aspects of the process. TDD/BDD methodology (Test-Driven Development — the development based on the testing, Behavior-Driven Development — the development based on the functioning) demands that tests have been written before the realization of services: in this case they play a role of the acceptance tests. As soon as the acceptance tests are in a process of carrying out, Test Task can be considered as executed. Defects in Test Task and errors of realization, revealed during testing, have to find their reflection in new tests. The regression tests, which check the lack of concrete errors, guarantee that these errors will not arise again in the future.

Any changes in the existing functionality of the system have to be accompanied by the start of tests. Usually it is provided automatically by means of the server of the constant integration. In spite of the fact that successful execution of tests does not guarantee absolutely correct work of the system (as a set of tests can not cover all situations), good tests give an acceptable idea of the system reliability.

Integration tests tools

Integration test can be found in the dictionary «Integration tests»Integration tests", which contains texts as for the platform so for configuration:

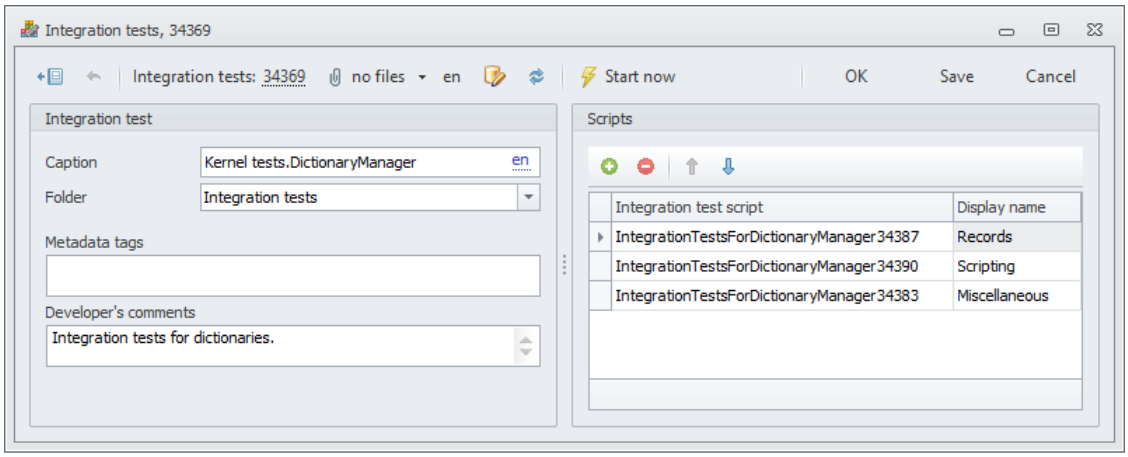




Dictionary window is divided into two parts: test groups tree is displayed on the left side, and text list of selected from left group is on the right. The dictionary records can be filtered by *text name* (*Name*) and *Tags* (Tag).

By command *Execute all tests* of toolbar as homonym command of main menu can starts the executing of active commands (marked by flag *Active*) of integration tests.

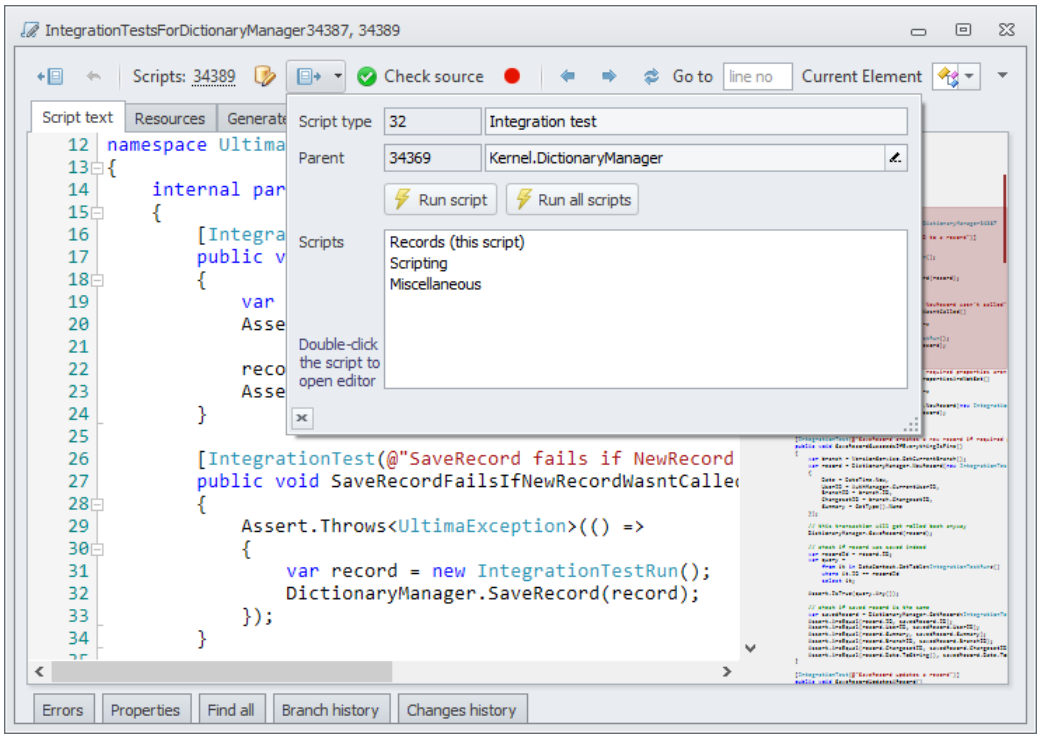
Each test contains one or more script (class), each script - one or more test-case (methods):



- *Caption* – text name;
- *Folder* – a group, the text belongs to;
- *Active* – tests with set flag will be executed by command *Execute all tests*;
- *Metadata tags* – tags used to describe text functionality;
- *Developer's comments* – comments of the application developer;
- *Scripts* – text script;
  - The scripts can be created or removed using corresponding buttons in the toolbar:
  - in case of script creation, edit form will be opened. Script will have on default the name *IntegrationTest* with numeral suffix;
  - you must specify the script name, which will be displayed in the test execution totals;in the field *Display name*
  - In case of removed script will be removed not only from the test, but also from directory Scripts;
  - The script can be opened in the edit form by double-click of the left mouse button on it in the list.
  - The script of the test selected in the edit form can be opened directly from the dictionary list form, having selected item *Edit script* in the context menu.
  - Script execution order is given by transferring of selected script by pressing and on the panel, execution order – from up to down;
  - Test-case execution order - methods inside script - also is from up to down.

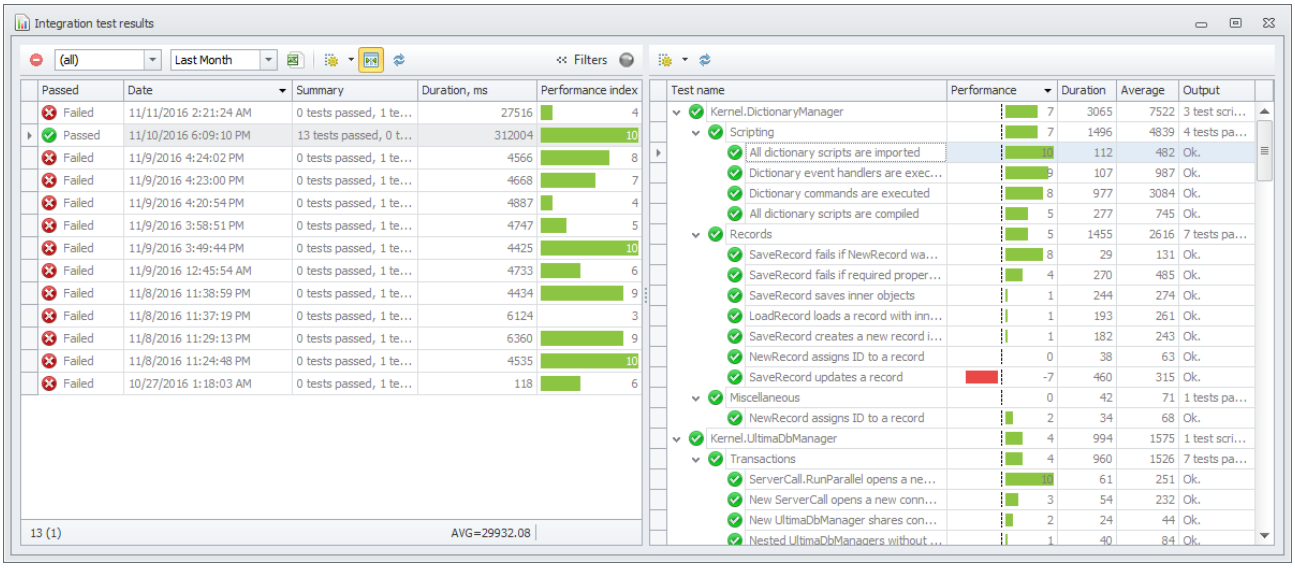
Test-cases usually writes of Arrange-Act-Assert pattern for standard unit-test: properties preparation, action, results checking. Thanks to the strict order of it execution the big business process can be divided into small stages, each of which will check separately.

Button Start Now on the tool panel of test editing form (also homonym paragraph of context menu of test list form) let execute it immediately. In the edit form of the script, buttons test button Run script (to run the current script) and Run all scripts (to run all scripts of the current integration test) are available:



As a rule, integration tests associate with data base. Do not influence to the real accounting the test executes in the separate transactions, which always recoiling. Each test (all scripts and cases) execute in one transaction, that is why several tests can be executed parallely in case package start.

The results of running tests are saved (and automatically opens upon completion) in the directory of the results of the «Integration test results»:



The directory window is divided into two parts: Integration test packages list is displayed on the left side, and results of selected from left tests in the tree structure is on the right.

Executed tests can be filtered by:

- , by which executed tests metadata version was marked (paragraph*(All)* – for all versions); Time of Integration test execution:
- the runtime of integration tests, for: Last week (*Last Week*), last month (*Last Month*), last year (*Last Year*), all dates (*Any Date*).

For executed integration tests the follows are shown in the list:

- *Date* – date and time of integration tests execution;
- *Summary – execution result:* Number of passed (tests passed) and failed (tests failed) integration tests;
- *VersionTag.Name* – tag, by which operated metadata version was marked integration tests;
- *Passed* – execution result:
  - Passed - all test-cases of all scripts was passed
  - Failed - at least one test-case of one integration test script was failed.

Also available are analytical columns, which allow evaluating the performance of launch:

- *Average duration- average runtime in milliseconds;*
- *Performance index — index of performance (ranging from -10 slowly to +10 quickly).*

You can easily enable conditional formatting via analytical columns in Data bars so that performance issues were immediately evident.

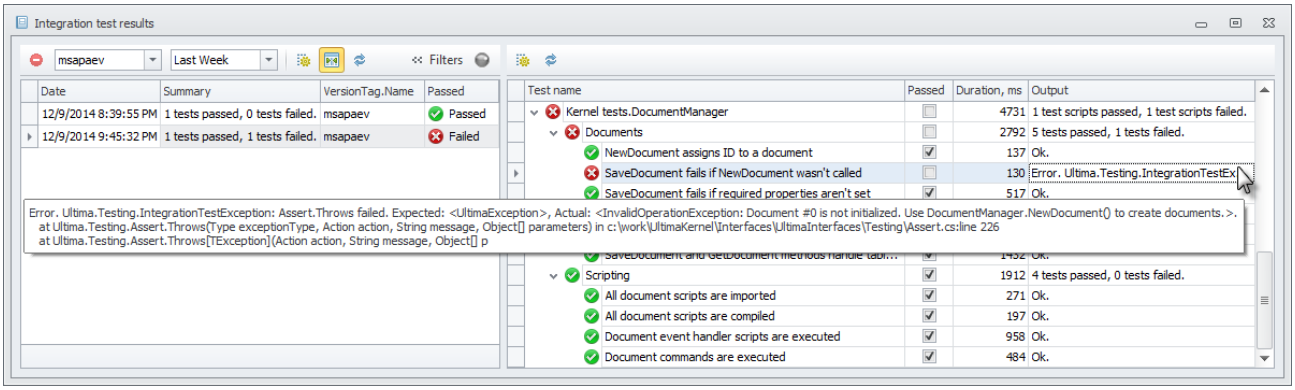
The results of performing the selected integration testsare displayed to the right in the tree structure: test cases are grouped into scripts, which, in turn, grouped by tests. Icon before the name shows the execution result: – if passed and – if failed. In case at least one test-case was failed, script execution result and all test also is consider as failed:

- *Test.Name* – name of integration test, script or test-case (script method);
- *Passed* – execution result - passed, in case flaf set;
- *Duration* – execution time in milliseconds;
- *Output* – execution result:
  - For integration test - the number of passed (*test scripts passed*) and failed (*test scripts failed*) scripts;
  - For script - the number of passed (*tests passed*) and failed(*tests failed*) test-cases;
  - For test-cases - passed (*Ok*) or failed(*Error*).

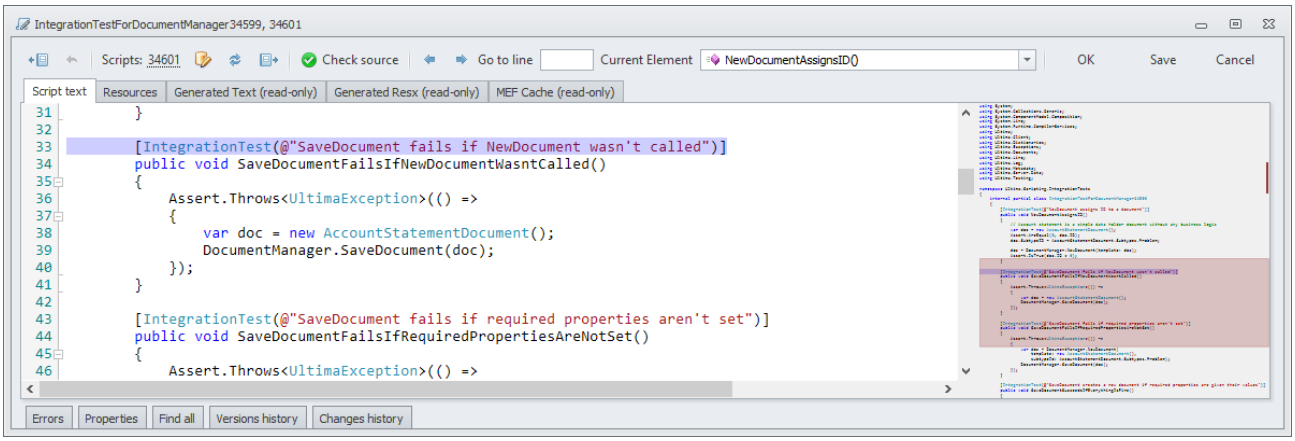
Also available in analytical columns, which allow evaluating the performance of a specific test result:

- *Average duration— average execution time of this test case in milliseconds;*
- *Performance index — index of performance (ranging from -10 slowly to +10 quickly).*

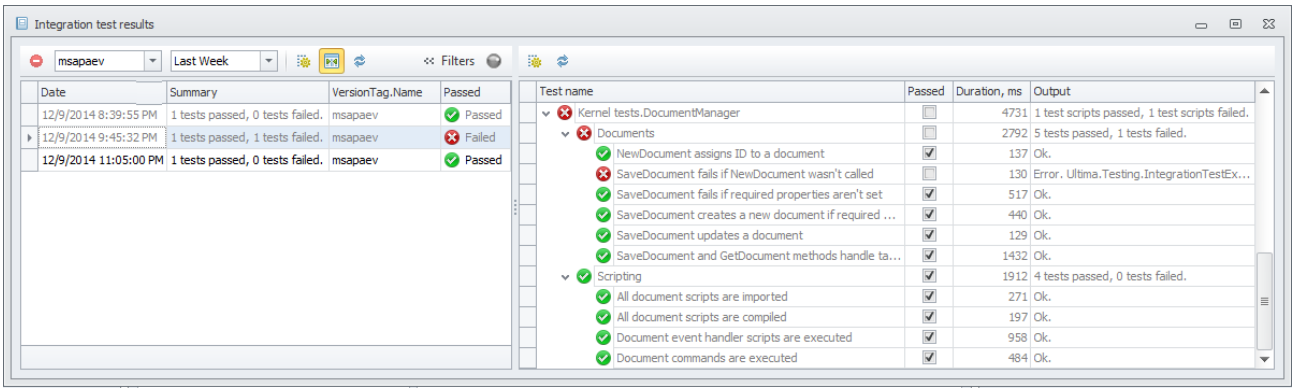
For failed test-case in the field *Output* the error text is also shows (the full text is displayed by cursor rollover):



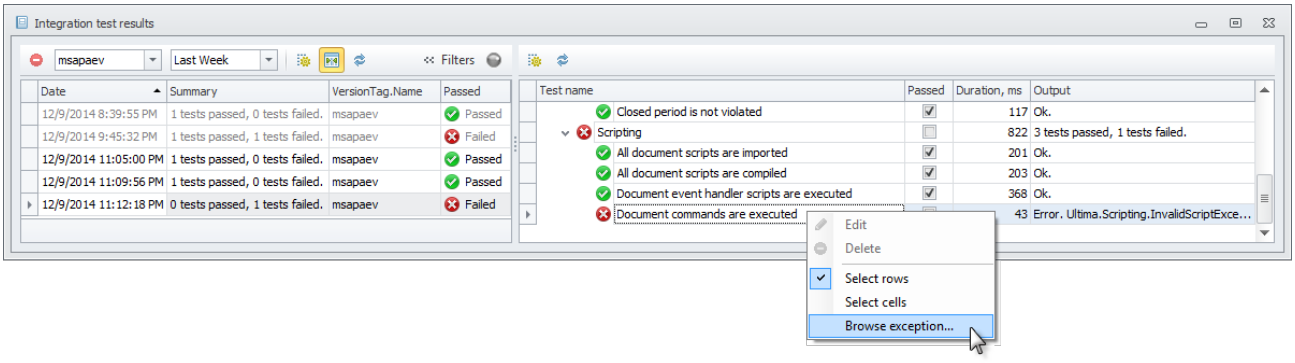
From test execution results you can get into integration test editor or into it script by double mouse left button click on corresponding result line. Double-click on result line of concrete test-case (script method) opens script editor on the need line:



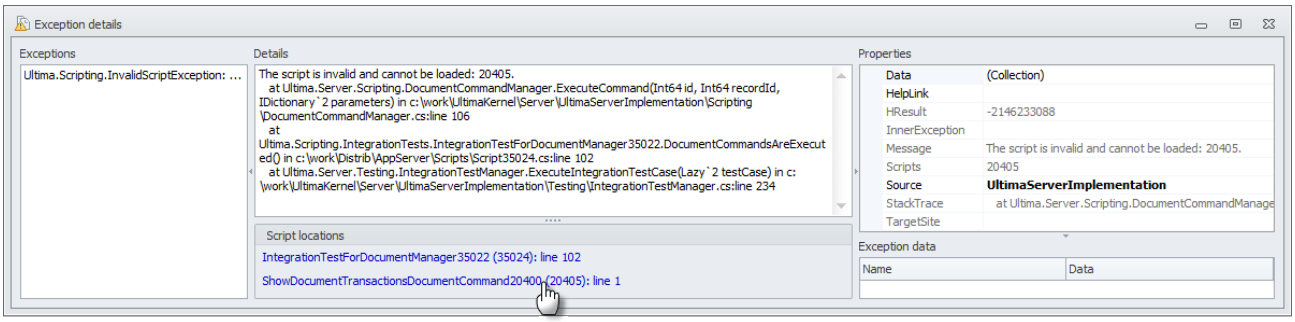
If the test executed on the metadata branch, marked by tag which differ from current branch, following message will be displayed when you trying to open script or test. Besides it, test results which aren't correspond to the current metadata version are shown in grey color:



For failed tests exclusions are automatically saved. This exclusions can be opened fro receiving detailed information (if the test had been executed in the different metadata version, the warning would be shown):



As exclusion, as usual, you have opportunity to get into concrete script line, which caused error:



You need to remember, that old exclusions can't be explore in this way - either exclusion can't be deserialize because of mismatch of metadata base, or line in StackTrace is not correspond to current script condition.

Integration server *TeamCity* execute tests suit at every distributive assembly. Error in test execution cause the assembly failed and all interested persons receive messages.

Client scripts

All types of scripts discussed previously are performed on the server side. In addition to them, the system supports several specific types of scripts that are performed on the client side:

- Dictionary editor scripts — allow including additional screen or business logic into standard forms of dictionary edit; replace or add new default controls;
- Documents editor scripts — provide similar opportunities for the standard forms of documents.

Dictionary editor scripts

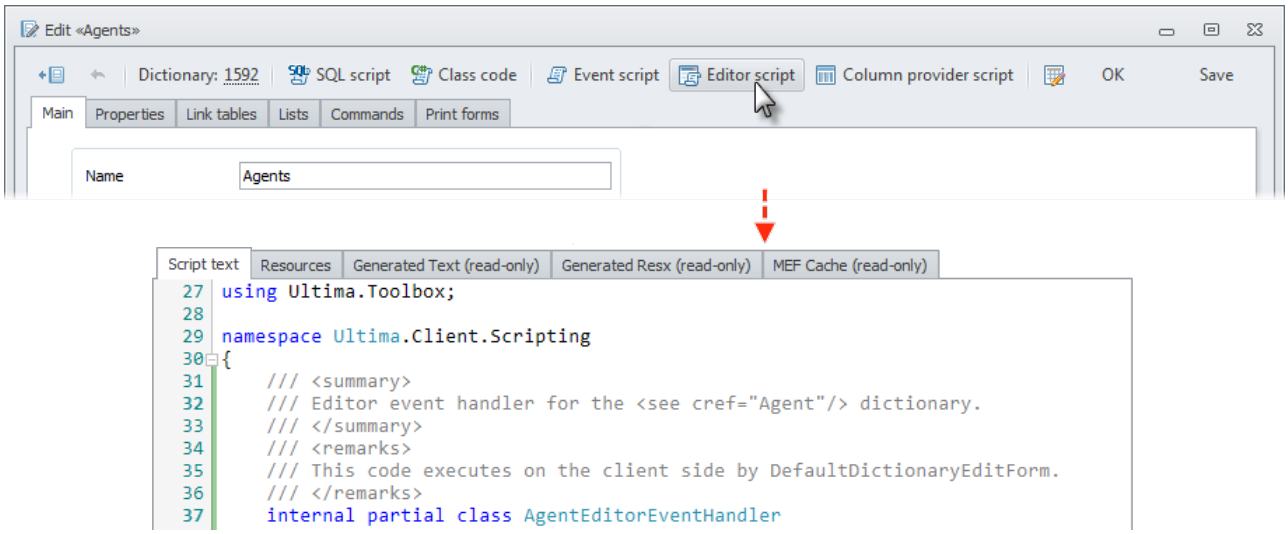
Functional of standard edit form satisfies most of the requirements to the usual directories. However, quite often there are reference books that to work correctly lacks some detail: additional validation on a text field, a non-standard element of management, filter or an event handler. Dictionary editor script can solve a lot of these small tasks, relieving the programmer from having to make the shape of the dictionary from scratch.

Editor scripts can solve the following issues:

- React to download events, creating and saving records.
- Load and display additional information in the form (map, schedule of rates and etc.).
- Block or hide fields depending on the record state or the current user's permissions.
- Change one field by changing the value of another field.
- Put filters on the field selection of the dictionaries.
- Create custom controls for fields, for example, Color picker or PictureBox.
- Combine records from multiple directories for collaborative editing.
- Get the information that is available only on the client (fingerprint scan, voice message recording).
- Fill defaults in nested tables, and dictionaries.
- Customize the list of displayed columns in the nested tables.
- Run validation values before sending to the server.

- Calculate on the wing such expressions as Amount = Price * Quantity, amount in rubles at the exchange rate and so on.

Handler of editing the reference book opens from a form of the reference book on the Editor script button:



Newly created handler text shows some possibilities offered by this type of scripts, providing them with necessary explanations. In particular, the new script adds to a form an element of management PictureBox. This element of management is added toLayoutControl, a visual representation of which can be set up directly on a running form.

If your processor adds additional controls in the form, and the form has already saved the layout, new elements will be hidden by default. In order to show them in the form, click the Layout Editor form and place new items on the right places. For forms without a saved layout new elements are displayed all at once, without further manipulation.

Multilingual installations layout system is stored separately for each language. If form has saved layouts for multiple languages, make sure that all items are relevant.

To get access to the script form controls, it is enough to declare the properties of the desired type and mark them with attribute *Control*, specifying the name of the editable properties as a parameter:

```
[Control("Name")]
private Control NameBox { get; set; }
```

Dictionary editor scripts implement *IDictionaryEventHandler* interface. The most important application properties and methods of a handler that can be used in a script:

- *RecordType* — edited record type;
- *ID* — edited record ID;
- *DataRecord* — edited record;
- *DisplayFormStatus(string text)* — displays the form status indicator (the same as displayed when executing a command or saving a record).

The handler may affect the behavior of the edit form by overriding the virtual method:

- *Attach* — allows attaching event handlers to a form.
  Instance of the form editor of the reference book is passed to the input *DefaultDictionaryEditForm.*
- *Detach* — allows you to disable the event handler attached to the method*Attach*;
- *AdjustStandardControl* — replaces the standard control for editing field by nonstandard.
  passed to the input:

- *controlHolder* — a container comprising a standard control for the field;
- *propertyDescriptor*, containing the description of the field for which the editor is needed;

- *CreateCustomControls* — allows you to add additional form controls.
  - list of control containers controlHolders is delivered at the input;

Please note: AdjustStandardControl and CreateCustomControlsmethodsare invoked during creation of elements of management of a form, therefore the properties marked with Control attributes at the time of their call aren't yet available. Accessing a control during the AdjustStandardControl method call can be done through the controlHolder parameter, having checked a property namepropertyDescriptor.Name.

Access to controls in these methods provides the ControlHolder wrapper class, which also provides properties placement of an element on an editing form: field name, data binding type, etc. If you are substituting a standard item, do not forget to specify the data bound property (e.g., controlHolder.BindingPropertyName = "Text").

The handler can react to events in the directory entry editing form by overriding the virtual methods:

- *BeforeCreate*— is performed before creation of record in reference book, after user presses the button of creation of a new record.
  - at the input parameters are passed to create a new entry that you can change or add to;
- *AfterCreate* — is performed after creating a directory entry, after pressing the record button to create a new record.
  - at the input parameters are delivered to create a new record, used to create a new record;
- *BeforeLoad* — is performed prior to loading of dictionary record to the form.
  - at the input a dictionary record code is delivered to be loaded;
- *AfterCreate* — is performed after opening a dictionary entry, but not in the case of a new entry. Using the handler, e.g. additional parameters can be loaded into the dictionary record.
  - At the input is delivered an opened dictionary record;
- *BeforeSave* — is performed before saving of dictionary entry, after the user has clicked a button for saving.
  - The saved dictionary record is delivered at input.
- *AfterSave* — is performed after saving of dictionary record.
  - The saved dictionary record is delivered at input.
- *AfterRejectChanges* — is performed after rejecting the user's changes made to the dictionary record.
- *DataRecordPropertyChanged* — is performed when you modify any property of dictionary record.
  - At the input a copy of *PropertyChangedEventArgs* is delivered with name of changed feature;
- *BeforeValidate* — is performed during validation ofdictionary record.
  - at the input parameters for creation of new record which can be changed or added are transferred;
- *Modified* — is called to check whether the dictionary record is modified after the last saving.

Please note: the majority of methods of the processor return theTasktype that the processor had an opportunity to call remote services (remote calls to the customer service is always asynchronous). If the method doesn't use remote services and doesn't call other asynchronous methods, in it it will be necessary to return Task.CompletedTask.

Example text editor directory handler Brand, which was created from a template:

```
namespace Ultima.Client.Scripting
{
    /// <summary>
    /// Editor event handler for the <see cref="Brand"/> dictionary.
    /// </summary>
    /// <remarks>
    /// This code executes on the client side by DefaultDictionaryEditForm.
```

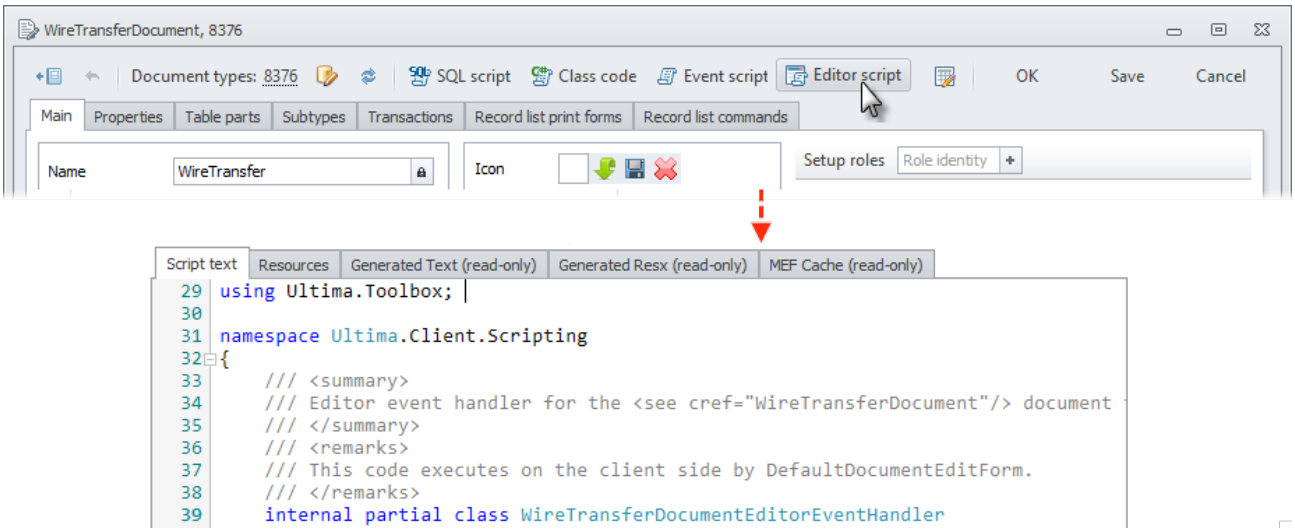
```
    /// </remarks>
    internal partial class BrandEditorEventHandler
    {
        // Declare properties for all controls you need to use.
        // Decorate properties with Control("PropertyName") attributes.
        [Control("Name")]
        private Control NameBox { get; set; }

        protected override void AdjustStandardControl(ControlHolder controlHolder,
IBaseDescriptor pd)
        {
            // Tweak or replace the default controls here:
            if (pd.Name == "MySamplePropertyName")
            {
                controlHolder.Control = new TextBox();
                controlHolder.BindingPropertyName = "Text";
            }
        }

        protected override void CreateCustomControls(List<ControlHolder> controlHolders)
        {
            // Create custom controls and bind them to the data here:
            controlHolders.Add(new ControlHolder
            {
                Control = new PictureBox
                {
                    Name = "XkcdPictureBox",
                    ImageLocation = "http://imgs.xkcd.com/comics/self_description.png",
                },
                ShowCaption = false,
                ItemName = "XkcdPictureItem"
            });
        }

        protected override Task BeforeCreate(IDictionary<string, object> parameters)
        {
            // Supply default values for the new record here:
            parameters["Name"] = Environment.MachineName;
            return Task.CompletedTask;
        }

        protected override async Task AfterLoad(Brand dataRecord)
        {
            // Load additional data to display by your custom controls here:
            await Task.Yield();
        }

        // The following virtual methods and properties are also supported:
        // protected override Task AfterCreate(IDictionary<string, object> parameters)
        // protected override Task BeforeLoad(long recordId)
        // protected override Task BeforeSave(Brand dataRecord)
        // protected override Task AfterSave(Brand dataRecord)
        // protected override Task BeforeValidate(Brand dataRecord,
ValidationErrorCollection errors)
        // protected override Task DataRecordPropertyChanged(PropertyChangedEventArgs e)
        // protected override Task AfterRejectChanges()
        // protected override bool Modified { get; }
    }
}
```

Please note: client scripts are run on the client, but stored and processed on the server. Therefore, the server must have access to all assemblies that are referenced by a client script. If your script is behaving strangely (no errors while saving, but the MEF-cache is empty and the script is not loaded on the client), most likely thing is that the server can not find all the necessary dependencies. Make sure that in the file of configuration of assembly probing path server includes the ThirdParty and Clientfolders with subfolders:

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <probing
privatePath="ThirdParty;ThirdParty/DevExpress;Client;Client/ClientModules;Client/C
lientModules/Base"/>
  </assemblyBinding>
</runtime>
```

Document editor scripts

This type of scripts is completely similar to dictionary editor scripts. The only difference is that all methods take in parameters not a directory record, but a document. Document editor script is opened from a document form by button Editor script:



The script has exactly the same features and methods as dictionary editor script. Here is the full text of the document handler WireTransfer, created from a template:

```
namespace Ultima.Client.Scripting
{
	/// <summary>
	/// Editor event handler for the <see cref="WireTransferDocument"/> document type.
	/// </summary>
	/// <remarks>
	/// This code executes on the client side by DefaultDocumentEditForm.
	/// </remarks>
	internal partial class WireTransferDocumentEditorEventHandler
	{
		// Declare properties for all controls you need to use.
		// Decorate properties with Control("PropertyName") attributes.
		[Control("SourceAccountID")]
		private Control SourceAccountBox { get; set; }
```

```
		protected override void AdjustStandardControl(ControlHolder controlHolder,
IBaseDescriptor pd)
		{
			// Tweak or replace the default controls here:
			if (pd.Name == "MySamplePropertyName")
			{
				controlHolder.Control = new TextBox();
				controlHolder.BindingPropertyName = "Text";
			}
		}

		protected override void CreateCustomControls(List<ControlHolder>
controlHolders)
		{
			// Create custom controls and bind them to the data here:
			controlHolders.Add(new ControlHolder
			{
				Control = new PictureBox
				{
					Name = "XkcdPictureBox",
					ImageLocation =
"http://imgs.xkcd.com/comics/self_description.png",
				},
				ShowCaption = false,
				ItemName = "XkcdPictureItem"
			});
		}

		protected override Task BeforeCreate(IDictionary<string, object> parameters)
		{
			// Supply default values for the new document here:
			parameters["Comments"] = Environment.MachineName;
			return Task.CompletedTask;
		}

		protected override async Task AfterLoad(WireTransferDocument document)
		{
			// Load additional data to display by your custom controls here:
			await Task.Yield();
		}

		// The following virtual methods and properties are also supported:
		// protected override Task AfterCreate(IDictionary<string, object>
parameters)
		// protected override Task BeforeLoad(long documentId)
		// protected override Task BeforeSave(WireTransferDocument document)
		// protected override Task AfterSave(WireTransferDocument document)
		// protected override Task BeforeValidate(WireTransferDocument document,
ValidationErrorCollection errors)
		// protected override Task DataRecordPropertyChanged(PropertyChangedEventArgs
e)
		// protected override Task AfterRejectChanges()
		// protected override bool Modified { get; }
	}
}
```

Update of script execution status

For the scripts, which execution takes significant time, it is correct to provide the user with information about execution progress.

For that purpose, the handler can update the text displayed in the client in the list of executed operations by filling in the field *ServerCall.CurrentCall.Text*. The value is added after the progress text:

```
using (MainForm.DisplayProgress("Progress description"))
```

If inside one group, set with *DisplayProgress*, several operations are executed in asynchronous manner, their values may overlap each other. In this case, they are recommended to split into separate groups.

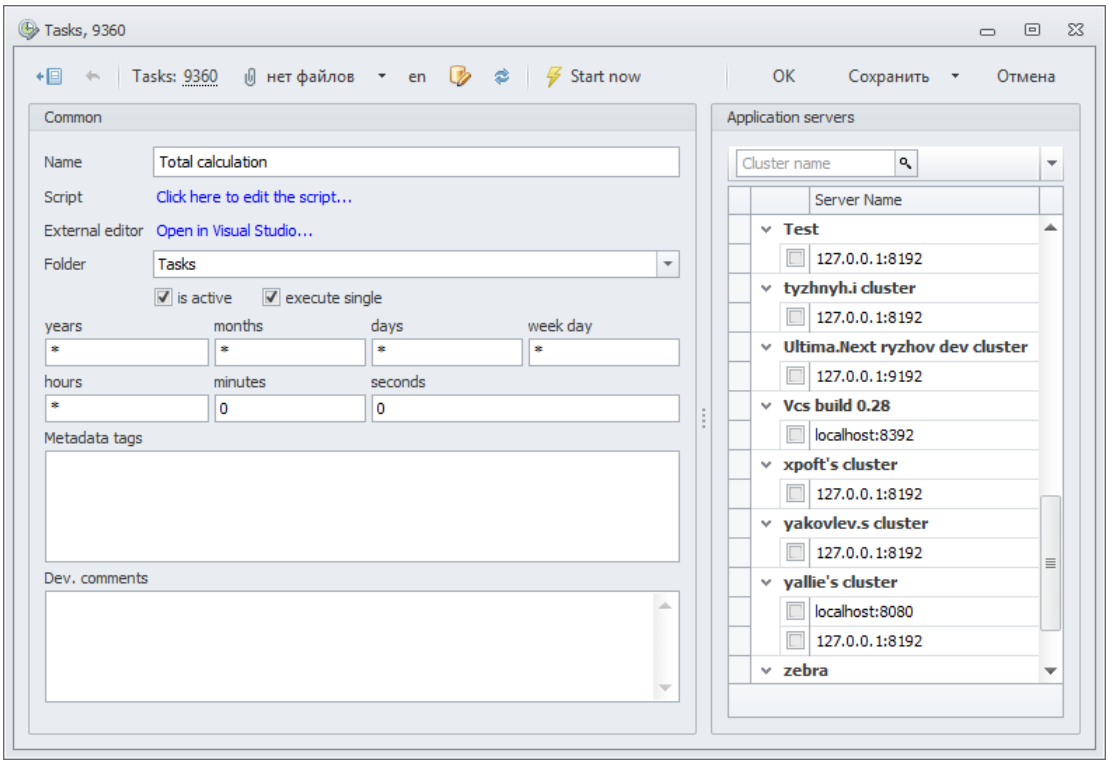
Script Updated Across Clusters

Development of the configuration is quite often conducted on the servers not united in a cluster. In this case it is possible that when a developer publishes his script changes to the main branch, and the main server does not receive the notification. In order to the main server can detect and use the script changes, being in the other cluster, it every minute polls the office log of scripts changes. If to start task scheduler on the server it is possible to allow this poll.

Journal of scripts changes represents a linking table ScriptUpdates, which keeps the date the last change on each script at all branches. The table is automatically updated during the editing of scripts and merge of changes. Finding an update on its branch, the application server dumps the cached version of the script, and at the next call to the script compiles it again. Due to the journal of changes the developer can conduct completion of scripts on the separate application server, which is not connected to the working cluster.

External script editor support

Built-in script editor is quite practical and is readily available on any client machine. Nevertheless it's still inferior to the full-featured integrated development environments such as Visual Studio. Many developers lack their accustomed tools, that's why the system can integrate with external script editors. Base solution has a ready-to-use VSScripting module for the Visual Studio integration. When external editor support module is loaded all scripted objects get an additional «Open in external editor» or «Open in Visual Studio» command link (external editor name for the links is provided by the integration module):



Supporting more external editors requires either upgrading the VSScripting module or writing a custom one. Integrating such a module with the client is easy: all it takes is to implement a simple interface allowing the system to interact with the module. The interaction is limited to the «open script in external editor» command:

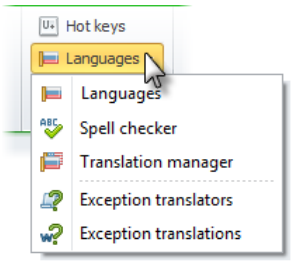
```
/// <summary>
/// Interface for external script editors such as Visual Studio.
/// </summary>
public interface IExternalScriptEditor
{
	/// <summary>
	/// Opens the specified script for editing asynchronously.
	/// </summary>
	/// <param name="scriptId">Script identity.</param>
	Task EditScriptAsync(long scriptId);

	/// <summary>
	/// Gets the name of the external script editor, i.e.: Visual Studio.
	/// </summary>
	string EditorName { get; }
}
```

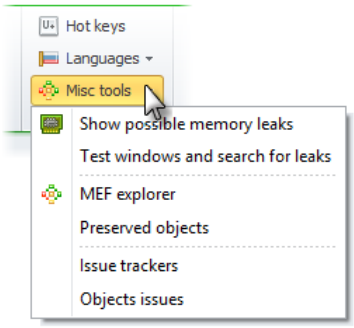
The module communicates with an external editor application (such as Visual Studio), creates temporary files on local disk and optionally provides additional commands. Editor integration module can interact with the client application just like any normal client module: open child forms, invoke application server's remote methods, etc.

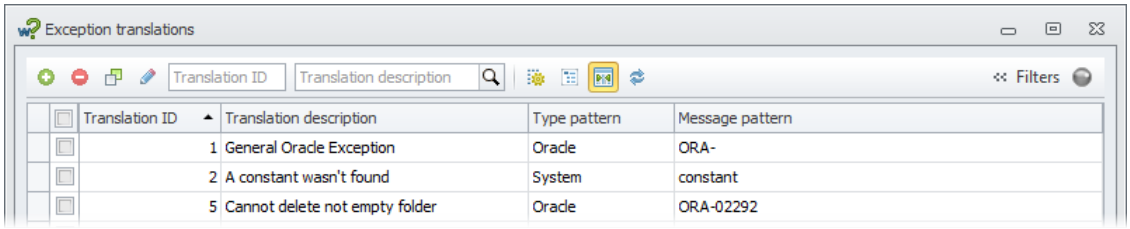
Note that the system only supports one external editor integration module at a time, so the custom editor integration module cannot be used alongside with the built-in VSScripting module.

Translation of exceptions

An exception (errors) thrown by the system can be set out in plain language. Viewing existing and creating new localized values (translations) of exceptions can be made in the dictionary Exception translations.

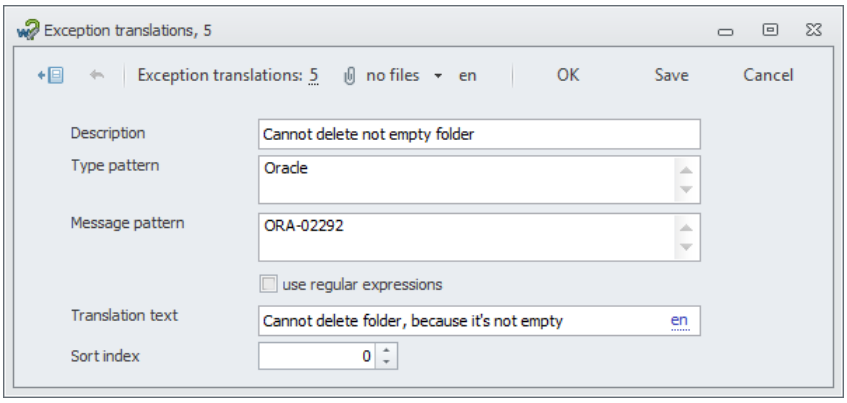






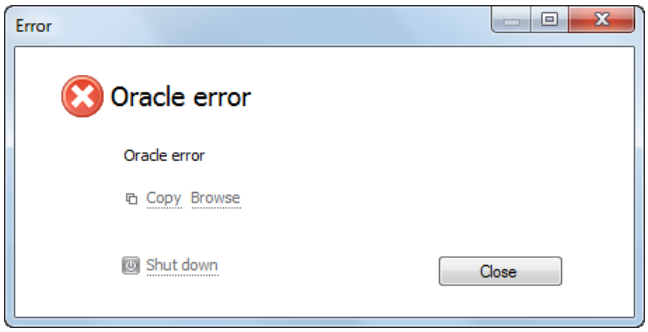
The dictionary records can be filtered by *Translation description* (*Translation description*).

The localized exception has the following properties:

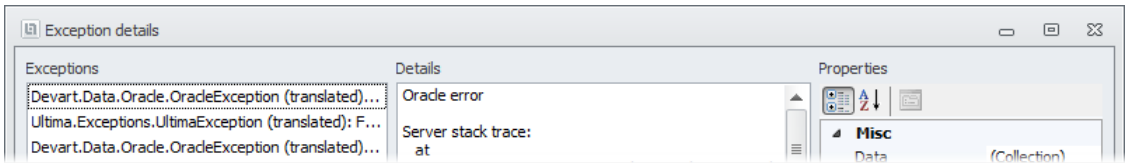


- *Description* – exception description;
- *Type pattern* – type of original exception;
- *Message pattern* – text of original exception;
- *use regular expressions* – a flag set if regular expression is used as the value of *Message pattern* property (detailed description of the interface can be found on MSDN website ⇨ eng/rus);
- *Translation text* – text of exception, will be thrown instead of original one;
- *Sort index* – sort index. For two localizations of similar exceptions, in which corresponding values *Type pattern* and *Message pattern* coincide, a localization will be used with large value of the index.

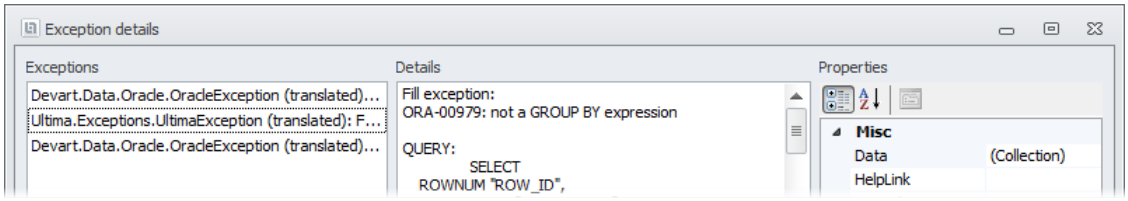
Let us consider localization of the exception by the example of error, which occurs during construction of the report:



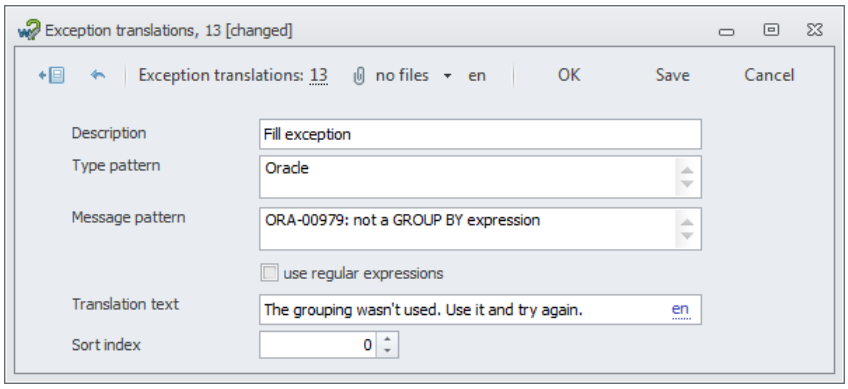
The information can be obtained from the exception details about its type (*Type pattern*) – this is Oracle error:



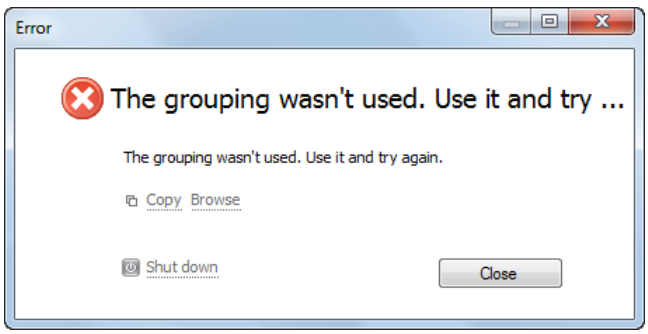
And exception text (*Message pattern*):



Using the obtained data, we create translation for this exception:

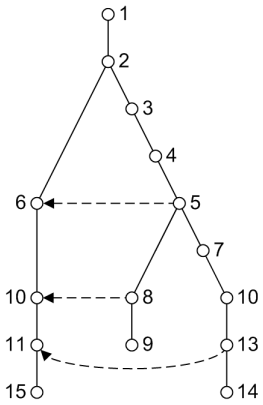


After saving of the translation, the next exception thrown by the system during repetition of actions, which resulted into the previous one, will look as follows:



Version control

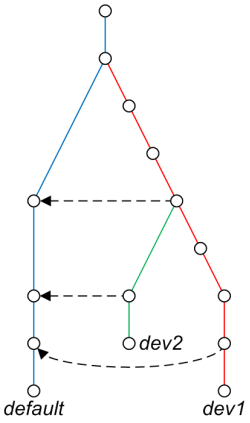
Ultimate AEGIS® system supports metadata versioning, which allows developing a configuration without affecting the work of ordinary users. The branches create isolation, while the complete graph of the system versions can be represented in a tree-like structure.



Each branch is virtually an exhaustive model, an off-line working copy of all metadata independent from other branches. One branch can be used for storing the actual production version of metadata, the other for developing, the third for giving a dry run to last changes, and so on. The branches can be synchronized with each other.

Each branch stores the full development history as a chain of commits. A commit is a set of changes supplied with additional information (author, date, comments and number of request, under which the changes were made). All sets of changes that belong to a particular branch are "read-only" except the last one, to which the current changes are being made. On the illustration, only the sets 9, 14 and 15 can be edited.

The same set of changes may belong to several branches. Firstly, when branches split off from each other, the entire history up to the branching point is common for them. Secondly, when synchronizing the branches, the commits are transferred from one branch to another. After the branches have been synchronized, all their metadata become identical, though the branches history is slightly different.



It is recommended to work with the versioning system in the following way. Each application programmer starts his own branch and work on it completely independent of other programmers. To synchronize all changes, programmers use the central branch "Default", which is always there in the system. The Default branch is considered the newest, the latest version of metadata and not usable so far.

To test changes before implementation, a branch called Test (or Stage) is created; this is used by testers, business analysts and certain end users responsible for acceptance. The branch syncs with the Default branch from time to time according to the plan of implementation of new features. If errors occur on this branch during the test, they are corrected directly on the branch; after that, the changes made are transferred to the Default branch and become available for syncing to all programmers.

A separate branch called "Production" is created for end users to work on. This branch syncs with the Test branch only, if, and only if, the functions of the latter has been carefully checked by testers, analysts and users responsible for acceptance. Any branch can be marked with a Read-only flag to prevent modifications. This is recommended to do only with the Production branch.

So, in addition to programmers branches and the *Default* branch, two more common branches are created in the system:

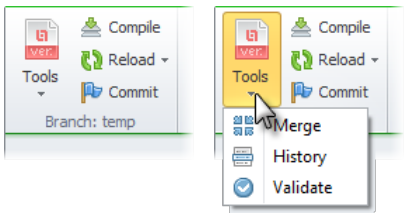
- *Production* – the latest stable version of metadata designed for ordinary users;
- *Test* – the version designed for testing before implementing.

The application server works with the metadata branch specified in the settings of its cluster. For example, for ordinary users, the *Production* branch should be specified in the cluster settings.

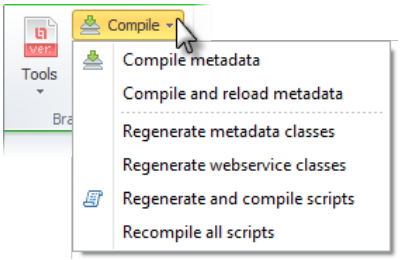
Thus, the versioning mechanism allows to:
- isolate metadata versions using the branches;
- branch and, afterwards, sync the branches, thus allowing each programmer to work in his own copy;
- store the entire history of metadata changes providing access to intermediate versions.

Versions tools

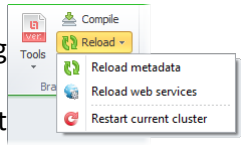
The tools, intended for work with a control system of versions, are located in the main menu in the tab “Developer”:



- *Branch: {Name}* – as the name of group the branch is specified, on which the current application server is started. It is a branch *temp* in the drawing;
- To create new branches it is necessary to use a history form of versions History, for more details see in appropriate section;
- Compile – list of actions is opened by clicking the key button of compilation:
  - Compile metadata – compilation of metadata, the result will be creation of binary files of metadata;
  - Compile and reload metadata – compilation and reloading of metadata;
  - Regenerate metadata classes – regeneration metadata classes;
  - Regenerate webservice classes – regeneration webservice classes;
  - Regenerate and compile scripts – regeneration and compilation of all scripts;
  - Recompile all scripts – recompilation of all scripts.



  The key button icon has two states:
  - a red button informs on existence of the changes made and not compiled yet in metadata;
  - a green button informs on relevance of the compiled library of metadata;
- Reload – list of actions is opened by clicking the key button of reloading:
  - Reload metadata – reset of metadata and modules (binary files), including web services;
  - Reload web services – reloading web services, at the same time before reset the web services will be automatically compiled;
  - Restart current cluster – reset of the current cluster of applications servers. The current server sends restart command to neighbors in a cluster, then it is restarted by itself. The command will wait for server restart.



- Commit – by clicking the key button the tool of fixing changes of the current version is started (if it is not fixed);
- Merge – the tool of versions merge of metadata is started when choosing a menu item;
- History – the view tool of the versions change history of metadata is started by clicking the key button;
- Validate – the tool of versions validation of metadata is started when choosing a menu item;

Besides, the dictionary of branches (*Branches*) is available in the “Administrator” tab of the main menu.

Recompile of scripts

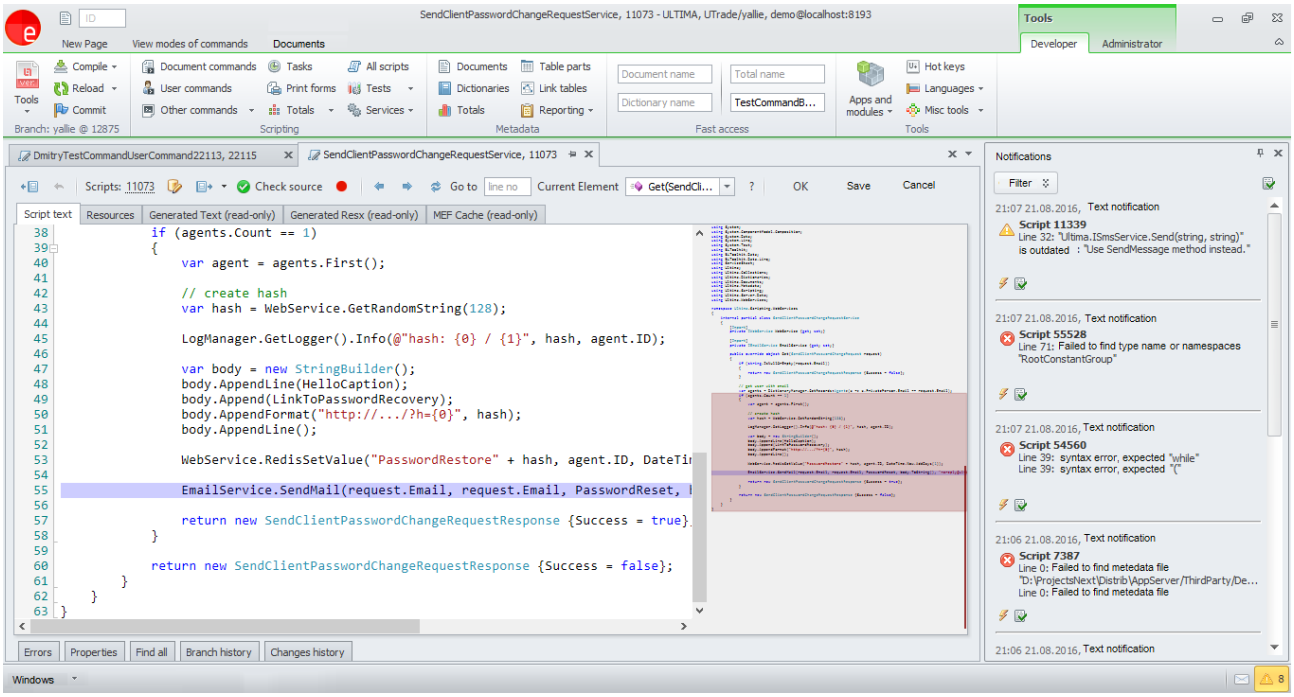
When major changes of metadata or services, as well as upgrading the kernel it is often required to check whether the existing scripts are broken. For such check there are two commands:

- Regenerate and compile scripts – regeneration and compilation of all scripts;
- Recompile all scripts – recompilation of all scripts (without generation).

The first command regenerate all scripts, updating of the generated part of the script, resources and MEF cache (see. script properties). After running this command there will be a lot of changes in the dictionary that need to be recorded. The programmer will be considered as an author of these changes, who executed the command Regenerate and recompile scripts.

The second command does not modify the scripts, but only checks that they are compiled. This command does not cause any side effects and do not leave any traces in the system.

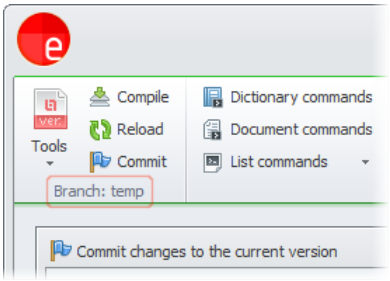
In the process of work the command for recompilation of scripts report all bugs via notifications:



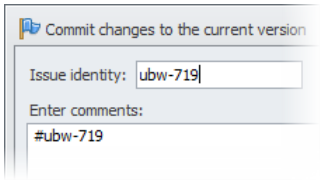
Notifications are sent in the course of command execution in real time to correct errors without waiting for the end of command execution. An action key button (⚡) allows opening a script on that line where the compilation error is found in each notice.

Commitment of changes to current version

Only the current version with the application server running can be committed. Editable versions with the committable changes are always located on the ends of branches. If the application server is running on a read-only branch, a commitment of current version will fail, and the command will be blocked. The current branch's name is displayed in the tab "Developer" within the name of a tools group of the version control system.



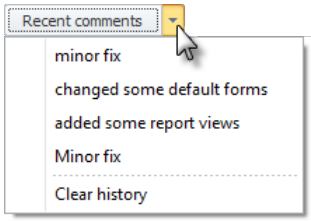
A form for commitment of changes consists of three parts. The area above is designed for input:



- *Issue identity* – ID of a tracker request, within the scope of which the committed changes were made (optional parameter). The ID may include digits, letters and special symbols and must match the request ID in the tracker URL; if so, the request can be opened in the tracker from a metadata object.

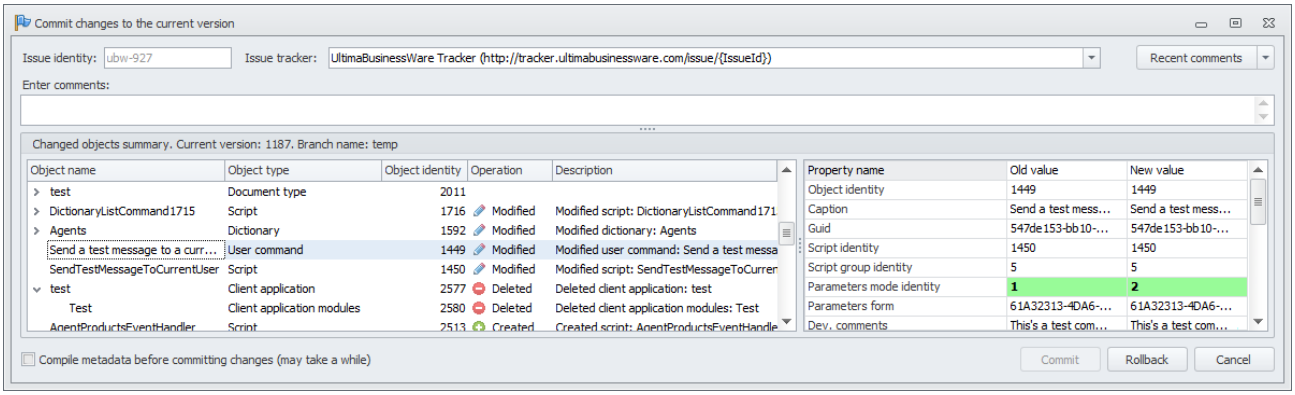
  When being entered, the request ID is automatically added to the comment field: if the comment field is empty, the ID is put at the start of the field; otherwise, it goes to the end of the field.

  All objects with the changes being committed will be marked as changed within the request scope. If changes made within the scope of two or more requests are being committed, there will be no way to correctly mark them with different IDs. Therefore, before proceeding to the next request, it is recommended to commit changes of the previous one.

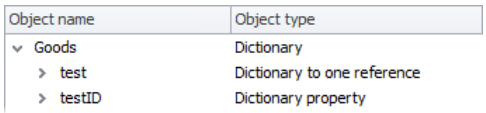


- *Issue tracker* – tracker, in the request scope of which the changes were made. A tracker should be specified, if a request ID *Issue identity* has been entered. The tracker that was selected during the latter commitment is entered into the field automatically;
- *Recent Comments* – by clicking the button one can select a recent comment entered by a user to the *Enter comments* field. *Clear history* clears the comments history;
- *Enter comments* – description of changes made to metadata; the field is mandatory.

The *Changed objects summary* area contains a list of metadata objects at the bottom, which were subject to changes. On the left, the changed objects are specified; on the right, the properties of the objects selected to the left are specified with detailed information on changes. The header of this area also contains the name of the current branch (*Branch name*):



In the list of the metadata changed, the objects are grouped according to their assignment: dictionary properties are enclosed into the dictionary, document subtypes into the document type, etc. Double-click a metadata object will allow proceeding to edit the object.



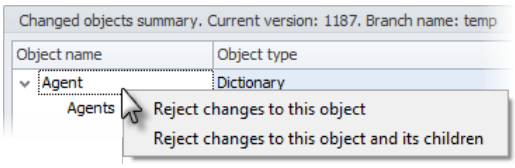
The list of objects changed contains the following information:

- *Object name* – metadata object name subjected to changes;
- *Object type* – metadata object type;
- *Object identity* – metadata object ID.
- *Operation* – operation that the metadata object was subjected to:
  - Created – object created;
  - Deleted – object deleted;
  - Modified – object modified;

- the operation may be missing if the metadata object was not modified directly, but the child object linked to it was modified:

Object name	Object type	Object identity	Operation
Order	Tablepart type	1980	
TestTablePartProperty	Table part to-one-reference	2445	Deleted

- *Description* – description of the change.



right-click on the object in the list of the metadata changed will open a context menu allowing to cancel the changes made (operation will be executed upon selection of the menu option):

- *Reject changes to this object* – cancel changes only for the object selected ;
- *Reject changes to this object and its children* – cancel changes for the object selected and its children objects.

Description of the metadata objects selected contains a list of all properties of the objects and their values before and after the change. Values different from each other will be highlighted in green:

- *Property name* – name of property of a metadata object that was subjected to changes;
- *Old value* – property value before the change;
- *New value* – property value after the change.

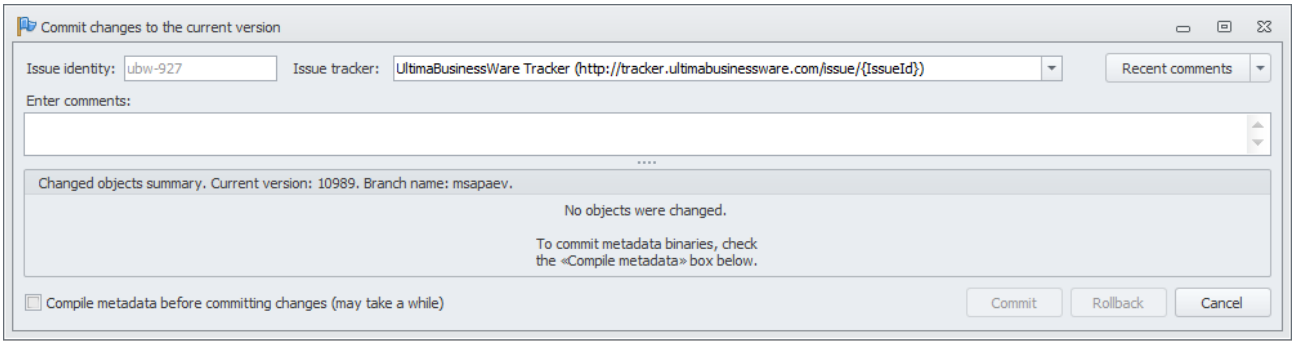
Checking the flag *Compile metadata* in the left bottom part of the form will allow compiling the metadata before their commitment. The compilation of metadata may require extra time. If what is being committed is an intermediate version not intended to be provided to end users to work with, there is no need to compile the metadata.

Click the Rollback button cancels all the changes made to the version. Upon click on the button, the changes will be rejected and the form for changes commitment closed. The metadata version will not be committed in this case.

The Commit button, which is used to commit changes, becomes available on entering the comment into the *Enter comments* field.

After the changes of the current version have been committed, a new version of metadata will be created, and the branch-tag of the current version will be automatically transferred to it. All changes made to metadata after the commitment operation, will be put into this new version.

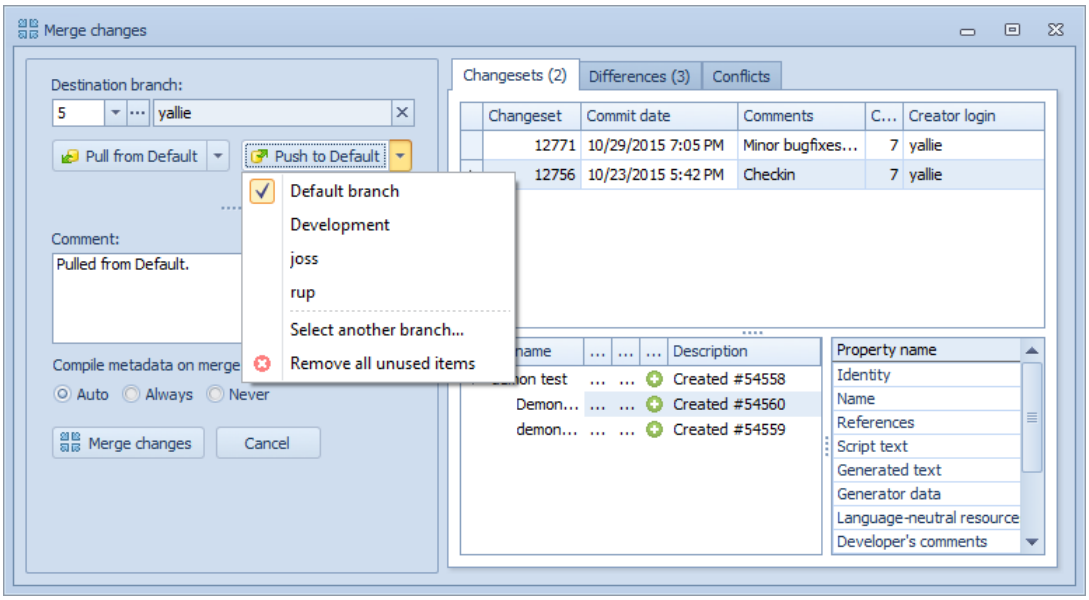
If no changes were made to the metadata version being committed, a warning will be shown: "No objects were changed". In the process, only binary files of the version can be committed; before this, compile the metadata (check the flag *Compile metadata before committing changes*):



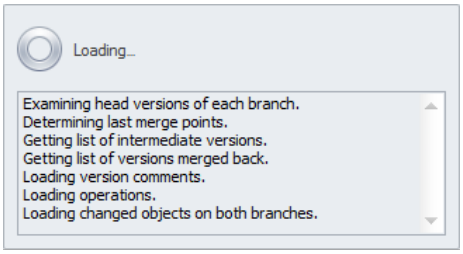
Merging of versions

Initially, the metadata versions tree of Ultimate AEGIS® system has a single branch marked with the branch-tag *Default*. The mechanism of versions merge is relevant in those cases when the tree of versions has several branches. Such situation is typical for a company with the staff number of more than one application developer. In this case, every developer conducts the development in his own branch marked with its own branch-tag, and the mechanism of versions merge is used for loading of changes from the main branch *Default* or pushing one's own changes into it.

The functionality of the versions merge form allows comparing metadata selected by branch-tag of the branch with the branch *Default* (or any other branch, if needed) and loading all changes, made after their forking (or after the last merge), into one of them:



- *Destination branch* – a branch-tag of configuration branch, into which the changes are pulled in from the *Default* branch (the header is changed to *Source branch* if the changes are not pulled in the selected branch but pushed from it into *Default* branch);
- *Pull from Default* – upon a click on the button, the changes will be loaded from the branch marked with *Default* tag, into selected branch. The information about the loading process is displayed in the pop-up window;

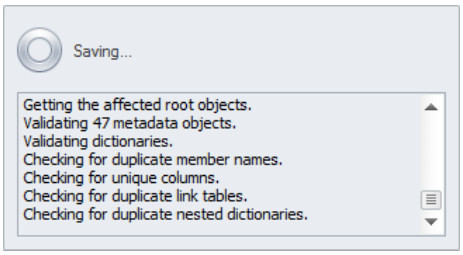


- *Push to Default*– upon a click on the button, the changes will be loaded from the selected branch into the branch marked with *Default* tag. The information about the loading process is displayed in the pop-up window;
- *Comment* – a comment, which the committed version will be marked with. When clicked on the button *Pull from Default* or *Push to Default*, a comment, which can be changed, will be inserted corresponding to selected action. The merge operation will not be performed if the comment has not been entered;
- *Compile metadata before committing changes* – a flag indicating the need in compilation of metadata before their commitment.
  - *Auto* – if checked, necessity to compile metadata is determined automatically. In this case, compilation is launched when changes are pushed into Default branch, if there are metadata or interfaces (incl. mobile interfaces) among the objects being loaded. In the process, the version containing no changes in metadata is not being compiled, but considered compiled. This allows loading changes in scripts without forcing the metadata collection to be reloaded;
  - *Always* – if checked, the metadata compilation process is always launched;

- *Never* – if checked, the metadata compilation process is never launched;

  Metadata compilation may require additional time. When changes are pushed into *Default* branch, it is recommended, if needed, to compile metadata (to check the flag *Auto*);

- *Merge changes* – upon a click on the button, the merge operation is carried out, and then the operation for branch version commitment, into which the changes are pulled in. The merge operation will not be carried out in case of conflicts not marked as resolved (*Resolved*). The information about the process of merge operation execution is displayed in the pop-up window;

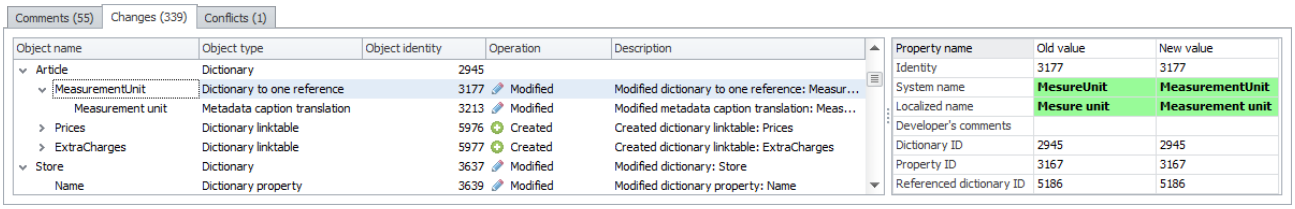


- *Cancel* – a click on the button closes the versions merge form;
- the information about loaded changes is grouped by tabs "Comments", "Changes" and "Conflicts".

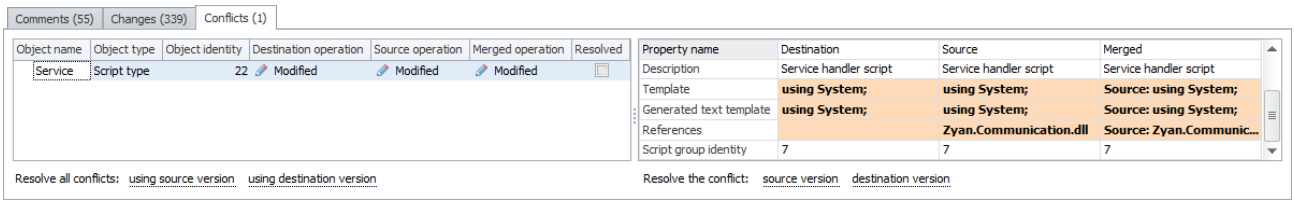
in the tab "Changesets", a list is given for all commits, from which the changes were loaded; a number of commits is indicated in the parentheses:

- *Changeset* – number of commit;
- *Commit date* – date of commit;
- *Comments* – comments entered during version commitment;
- *User identity* – ID of the user that committed this version;
- *Creator login* – name of user, who committed the version.

in the tab "Differences", a list of changes is given, which were loaded, a number of changes is indicated in the parenthesis. The list consists of two parts: on the left, the metadata objects are specified, which were subject to changes; on the right, the properties of the objects selected on the left are specified with detailed information about changes. Implementation of the list of changes and its functionality are similar to the changes in [the changes commitment form]:



in the tab "Conflicts", a list is given for all conflicts arisen, which the system is not able to resolve automatically. A number of conflicts is indicated in the parenthesis:



The conflicts are overlapping and at the same time different changes of one and the same objects in the merged versions, e.g.:

- editing of metadata objects in one of the merged versions and its deletion in another;
- the editions of metadata objects differing from each other, e.g. script code;
- creation of metadata object of one and the same type, e.g. user command, with similar name but different properties/parameters.

The conflict will not be represented for instance with:

- creation of new children objects of metadata, with a different name, for one and the same object, e.g. properties for the dictionary (document type, etc.). Both properties will be available in the version produced as a result of merge;
- deletion of one and the same object of metadata.

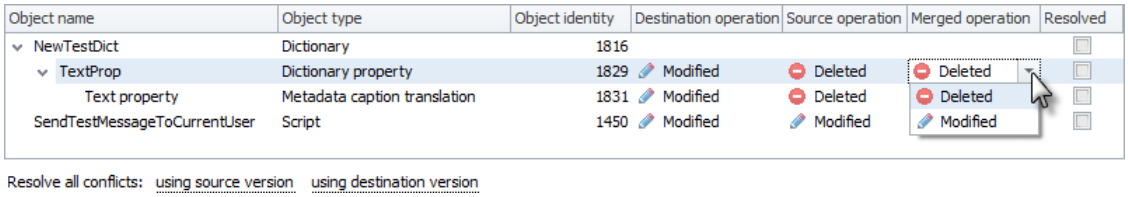
- identical change of the object on both branches. e. g., renaming a dictionary field.

A conflict arisen in case of changes overlapping should be resolved. It can be made only manually. In case of different operations for the object (the object is deleted in one version, and edited in another) – this is selection of actual operation. In case of different editions of one and the same object, when they are actual – this is selection of one edition (e.g. the most complicated and labour intensive), and subsequent manual introduction of ignored edition after merging. Conflicting script text can be merged using an external conflict resolution tool to get the combined script containing up-to-date changes from the both branches.

The permissions tab is divided into two parts: on the left, the metadata objects are specified, which changes in the merged versions overlap between each other; on the right, detailed description is provided for overlapping changes of the objects selected on the left.

The list of metadata objects, which changes are in conflict between each other, contain the following information:

- *Object name* – metadata object name. Names are grouped according to their assignment: dictionary properties are enclosed into the dictionary, the document subtypes are enclosed into the document type, etc.;
- *Object type* – metadata object type;
- *Object identity* – metadata object ID.
- *Destination operation* – an operation, which the metadata object is subject to in the version, where the changes were loaded from (it is marked with the tag selected in *Destination* element):
  - Created – the object was created;
  - Deleted – the object was deleted;
  - Modified – the object was modified;
  - the operation may be missing if the metadata object was not modified directly but the child object linked to it was modified;
- *Source operation* – an operation, which the metadata object is subject to in the version, which the changes are loaded into (it is marked with the tag selected in the *Source* element);
- *Merged operation* – one of two operations *Destination operation* or *Source operation*, which falls into the merged version. In the list being opened upon a click of the left mouse button on the operation, an operation of which version can be selected for leaving as a result of merging:



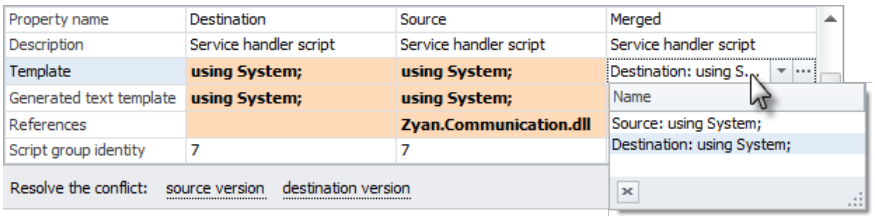
- *Resolved* – a flag, which the resolved conflicts are marked with. If at least one conflict is not marked with this flag, the merge operation will fail;
- *Resolve all conflicts* – the links allowing performing selection of all conflicting operations, which will fall into the merged version:
  - *using source version* – the conflicting changes from the version, marked with the tag selected in *Source* element, will fall in the merged version;
  - *using destination version* – the conflicting changes from the version, marked with the tag selected in *Destination*element, will fall in the merged version.

  When clicked on any of the link, all conflicts will be marked with *Resolved* flag as resolved.

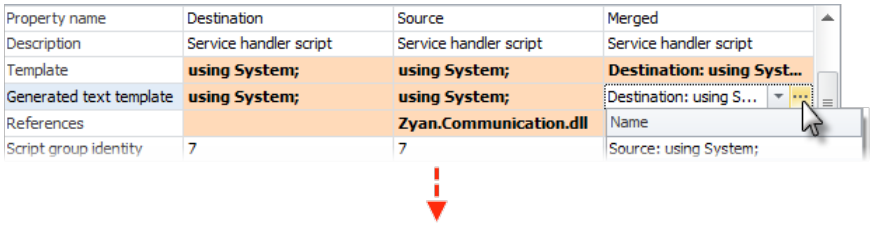
Detailed description of overlapping changes of selected metadata objects contain a list of all properties of the objects and their values in merged ones as well as in the resulting version. The changes, different from each other, will be highlighted orange pale:

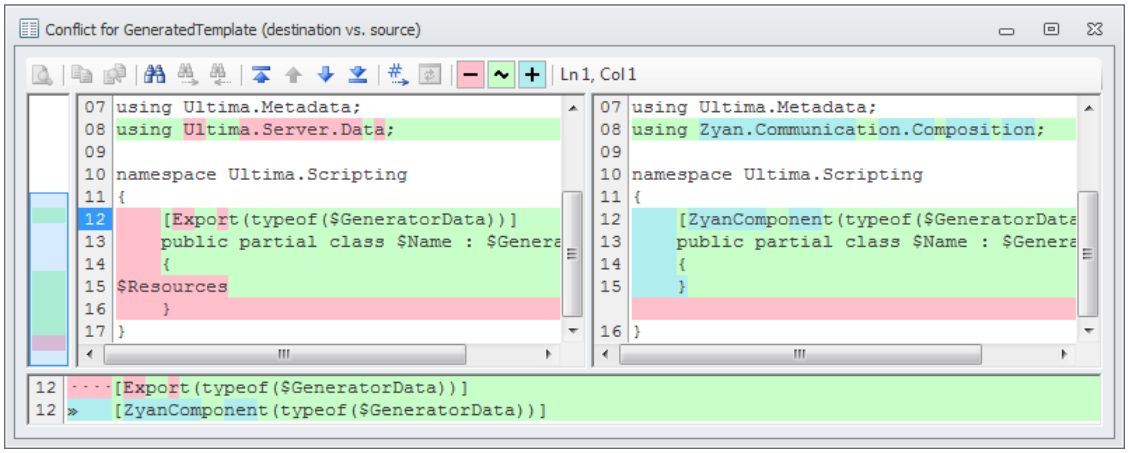
- *Property name* – a property name, which was subject to changes of metadata object;
- *Destination* – value of the property in the version, marked with the tag selected in *the From* element;

- *Source* – value of the property in the version, marked with the tag selected in the *To* element;
- *Merged* – one of two changes, which falls into the merged version. Its value is preceded with the text «*Source:*» or «*Destination:*» whichever is selected. In the list being opened upon a click of the left mouse button on the change, a value of which versions can be selected for leaving as a result of merging:



The conflicting text values of changed properties can be compared in a separate form opened by clicking ...:





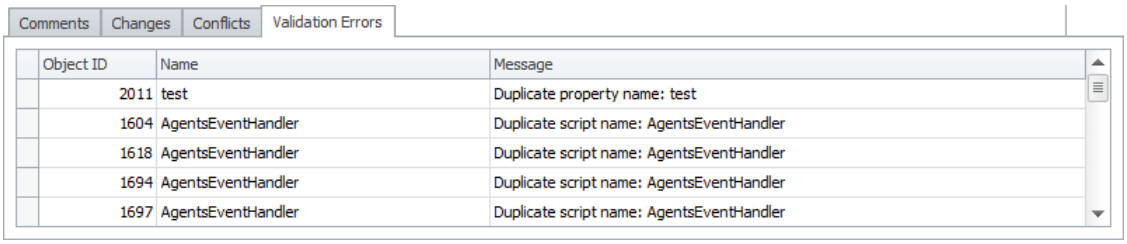
In the left part of the form for comparison of the texts of conflicting properties, a value is provided for *destination* property (version, marked with the tag selected in *Destination* element); in the right part of the form – a value for *source* property is provided (version, marked with the tag selected in *Source* element). Comparison of the texts is carried out in such a way as if the text *source* was produced by changing of the text *destination*:

- green marks the changed text;
- pink marks the deleted text;
- blue marks the added text.

In the lower part of the form, both versions of the row, on which a cursor is set, are provided for comparison;

- *Resolve all conflicts* – the links allowing performing selection of all values of conflicting changes, which will fall into the merged version:
  - *source version* – the values of conflicting changes from the version marked with the tag selected in *Source* element will fall into the merged version;
  - *destination version* – the values of conflicting changes from the version marked with the tag selected in *Destination* element will fall into the merged version.

After completion of merge operation, the metadata are checked for errors. After being found, such errors are displayed on tab "Validation Errors". A number of errors is indicated in the parenthesis:



The metadata version containing validation error cannot be compiled. Correspondingly, if during merging a flag of metadata compilation *Compile metadata before committing changes* was set, the merge will not be completed. The validation errors as well as the conflicts will have to be corrected manually too.

The list of validation errors contains the following information:

- *Object ID* – object ID;
- *Name* – a name of the object, which checking detected an error;
- *Object Type* – object type;
- *Message* – error message text.

By double click on the object it can be opened for editing. Validation is described also in the section Metadata error check.

Let us consider a sequence of actions for the most common situations of versions merge, when the developer requires pushing the changes made from own branch into the *Default* branch:

1. At first, the changes made to the last version of own branch should be committed. Firstly, the non-committed changes cannot be pushed into another branch. The merge will be executed only for previous-last committed version in the branch. Secondly, the changes from the other branch cannot be pulled into non-committed version.
2. After that the changes should be pulled in from *the Default* branch. It should be made because the changes could be already made by other application developer into *Default* branch after the last pushing of changes from current branch:
   - if pushing at first the changes of own version into the *Default* branch, resolution of potentially possible conflicts will fail, since it can be made only in the current version, which the application server is started from;
   - these updates will be necessary in any case since conducting collective development is simpler in relatively up-to-date version of metadata.

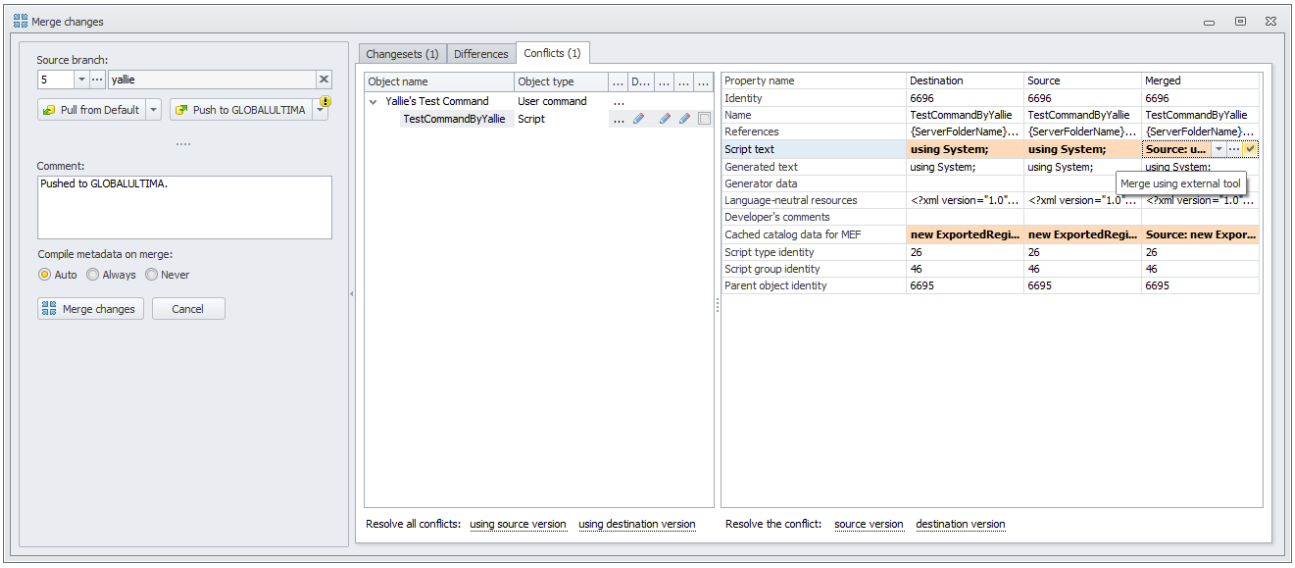
   If a warning appears that the non-committed changes are present in *Default* branch – the situation is unlikely but possible – they should be committed at first. After that the conflicts should be resolved and the pulled in changes with metadata of own branch should be merged. Moreover, the metadata version, which the changes were pulled into, are committed. Now the metadata of current branch differ from the metadata of *Default* branch only with the changes made by the application developer.
3. And finally, the changes from own branch into *Default* branch should be pushed through. If the other application developer has not managed to push there own changes after the loading at stage 2 of changes to *Default* branch – no conflicts will be present (if they managed to do it, the actions of the second stage will have to be repeated). Only merging will remain to complete the operation. In the process, the metadata version, to which the changes were pushed, are committed.

Thus, in order to push the changes from own branch to *Default* branch, performance of two subsequent merge operations is required.

Script text conflict resolution

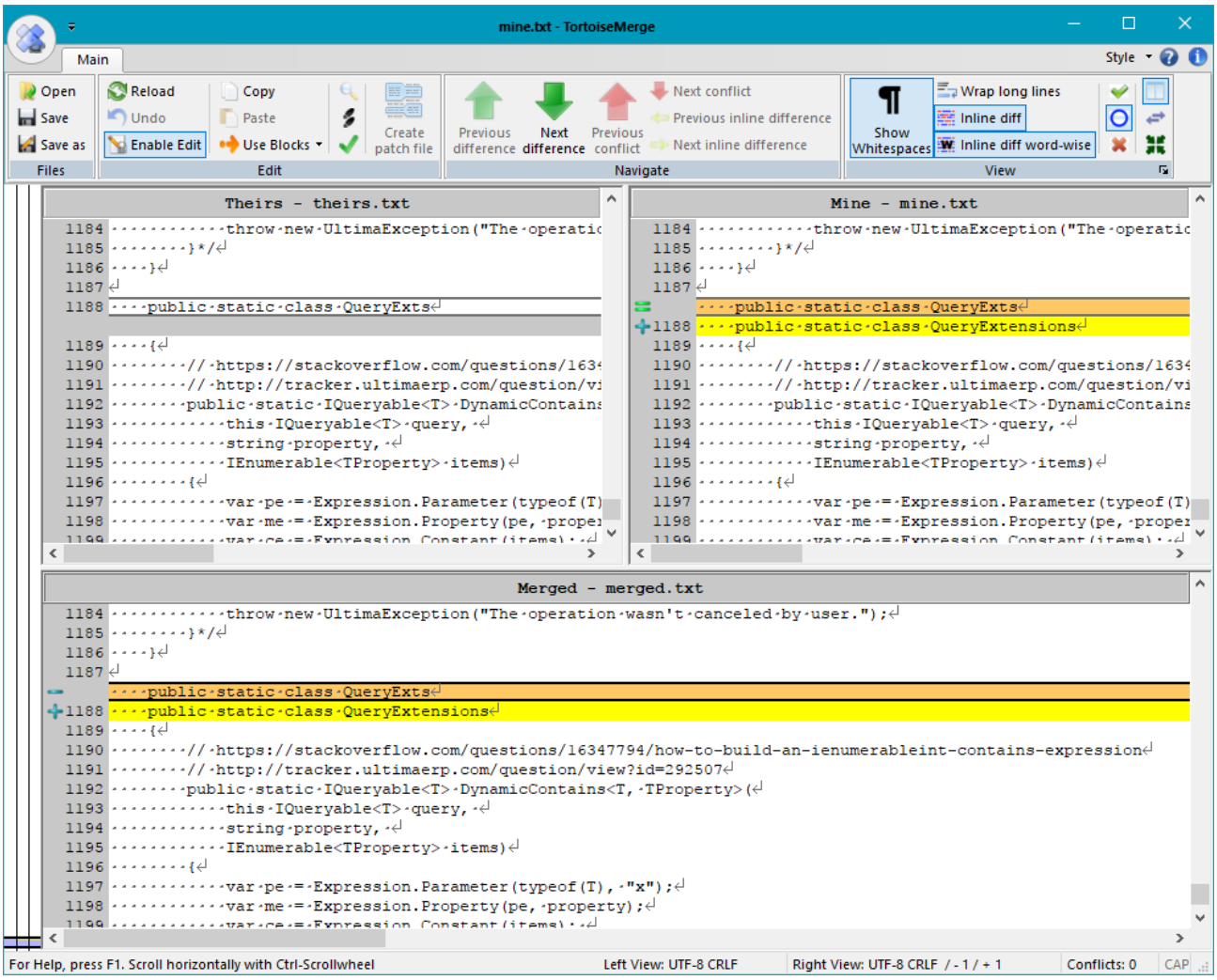
Merge conflicts are normally resolved by choosing one of the available versions. But merging scripts is more challenging. Consider a script modified by two developers on their own branches: the first developer adds a new method to the script, and the second developer writes another method, perhaps in a different part of the text. One cannot select either version as a merge result: both added methods need to be present in the resulting script text.

This important specific case of the conflict resolution requires a dedicated tool (we currently use external TortoiseMerge, a widely used free diff/merge utility). To resolve the text conflict using the external conflict resolution tool select the conflicting text field and click the «Merge using external tool» button:

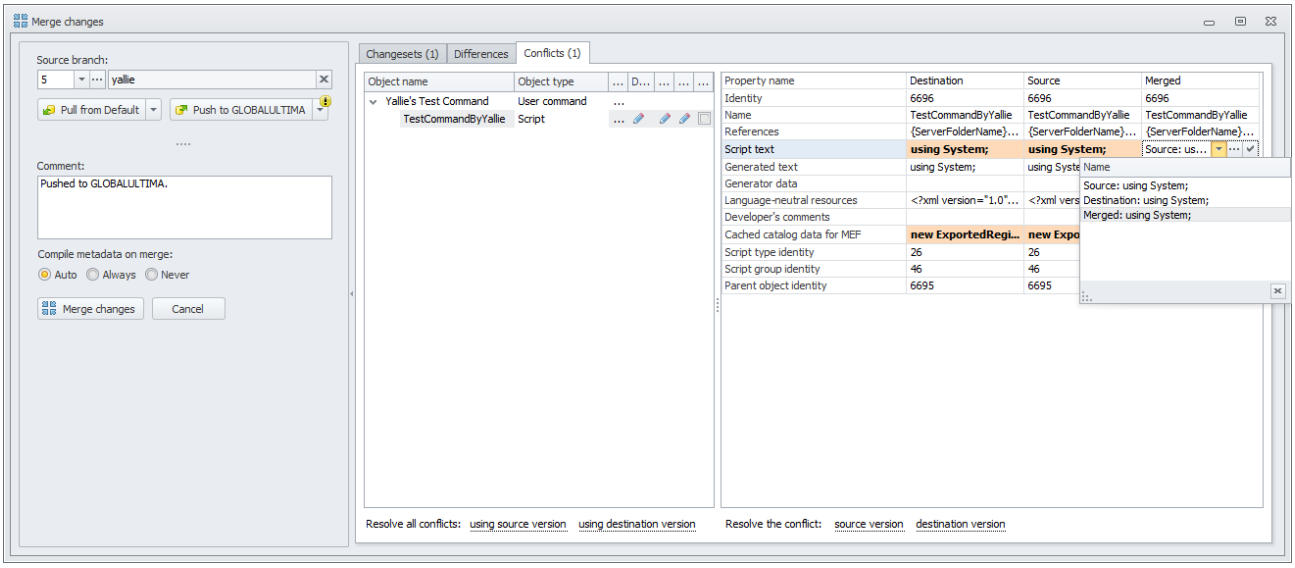


-

The system saves the conflicting script versions as temporary files on the local disk and launches an external merge tool. The merge tool displays three versions of the text: the upper left panel shows the first developer's version, the upper right panel has the second developer's one, and the lower panel displays the combined version, which is editable. This version merges changes made by both developers. The conflicting lines are highlighted in red. To complete the merge one needs to make sure that the combined version of the text has no conflicts highlighted in red. In the simplest case (if the lines changed by both developers are located in the different parts of the file and don't overlap) the merge tool is able to resolve the conflict automatically once all files are loaded. In that case the result can be saved as is, with no changes at all:



When the merged file is saved, the external tool is closed, and the temporary files are cleaned up. The merge result is added to the conflict resolution form. The drop-down list for the given text field now has a *Merged* item corresponding to the combined version of the script:

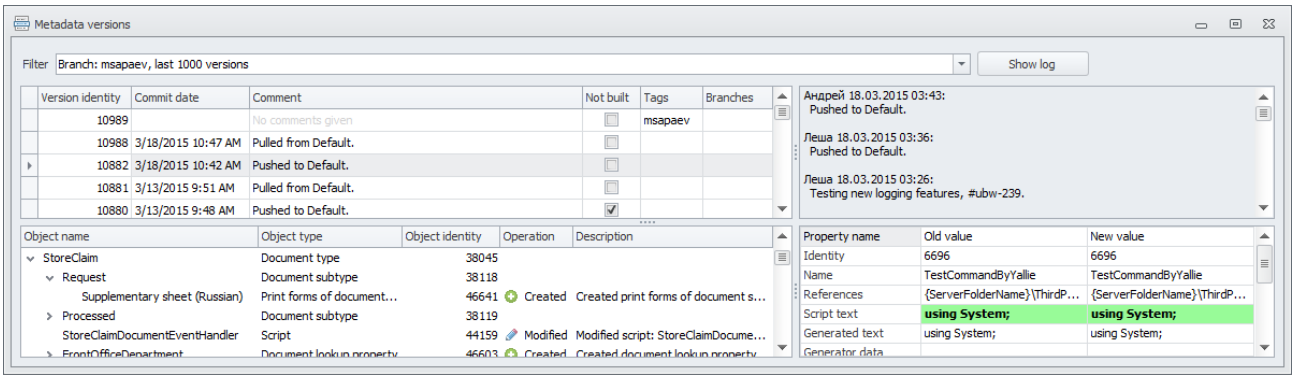


All left to do is to set the *Resolved* checkbox that marks the selected conflict as resolved, and the combined script version can be uploaded to the destination branch.

Version history

 The form of version history allows looking at all branch of versions to the first node for the selected branch-tag or version branch to the first node starting with the version marked with a normal tag.

By default when the form opening history of a current version is shown:



The history form is divided into four parts:

- at the upper left a list of branch versions marked by the selected tag is located;
- at the upper right – an user name, date of changes fixing and comment entered when changes fixing for the version selected at the left;
- at the lower left – a list of metadata objects undergone changes in the version selected at the upper right (it is similar to the list of changes in the form of changes fixing);
- at the lower right – a property list of metadata objects selected at the lower left with detailed information on changes (it is also similar to the property list in the form of changes fixing).

Branch history contains the following information:

- *Private* — a flag meaning that this set of changes will be passed in case of synchronization with other branches.
- *Changeset* – number of a change set.
- *Creator* — change author.
- *Committed* – date of a change set fixing;
- *Comments* – a comment entered during version commitment; Unrecorded version has a comment No comments given;
- *Source branch* – a source branch on which this set of changes appeared at the first time.

In the form of history viewing possibility of additional filtering is realized:

In addition to *Branch* metadata the following parameters filtering is available:

- *Issue identity* – a request identifier in a tracker if it was entered when changes fixing;
- *Object* – metadata object;
- *Show only this object operations* – the set flag allows showing in the list of the changed metadata objects at the lower left only changes of the selected *Object*;

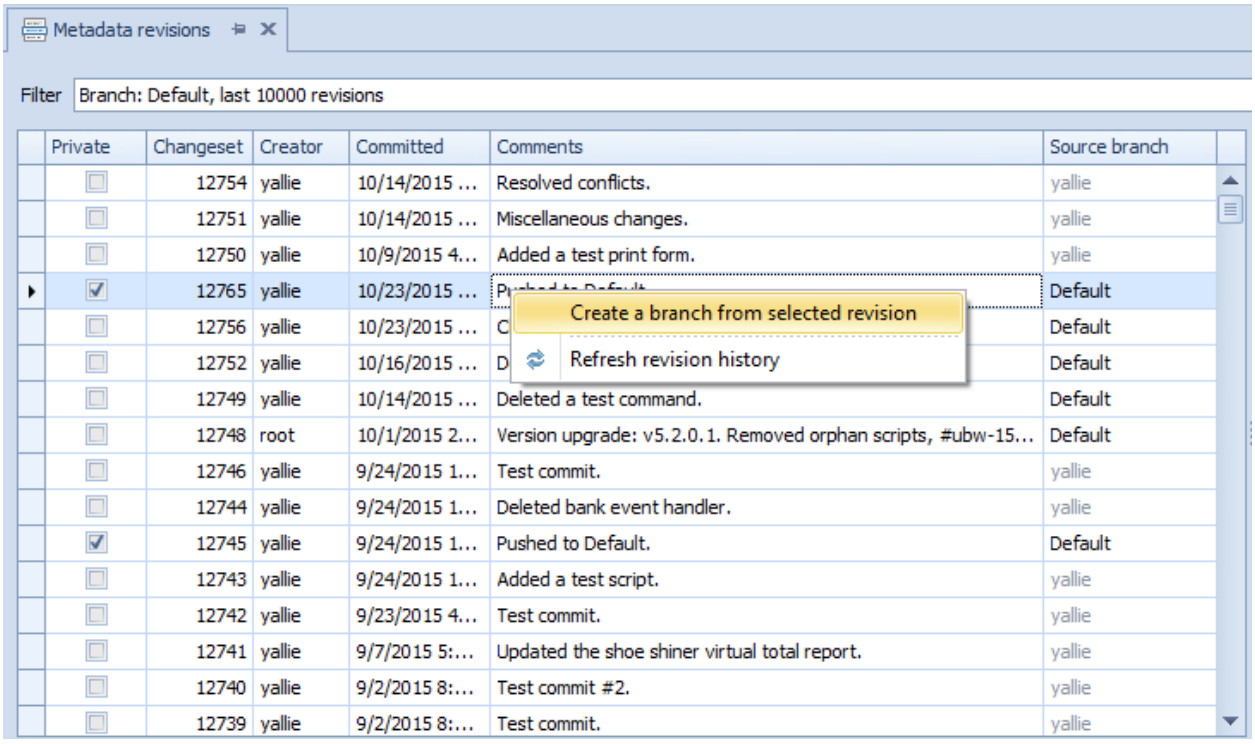
- *User* – the user login making the changes;
- *Period* – time slot when fixing was made;
- *Limit* – number of removed records (change sets).

For use of the filter click *Show log* to the right of it. The selected *Branch* and record *Limit* are remembered and automatically applied in case of the following opening of the history form.

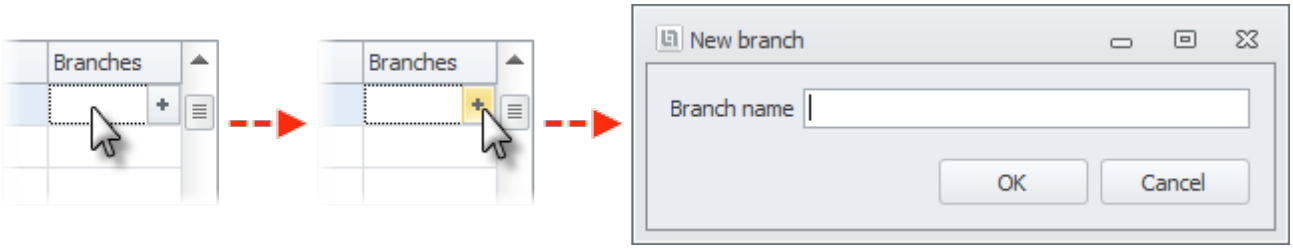
Creation of new metadata branches

The form of history viewing allows creating new branches by cloning of already existing. For this purpose:

1. In the history form to remove a branch history which should be cloned (usually it is Default branch). For this purpose select the appropriate branch from the filter.
2. Select revision from which create a new branch. Not all versions approach, it is connected to features of the internal version organization.
3. In the shortcut menu select the Create branch from selected revision command, or by means of the button + which appears in case of click in the Source branch field in a line of that revision which shall become the beginning of a new branch:



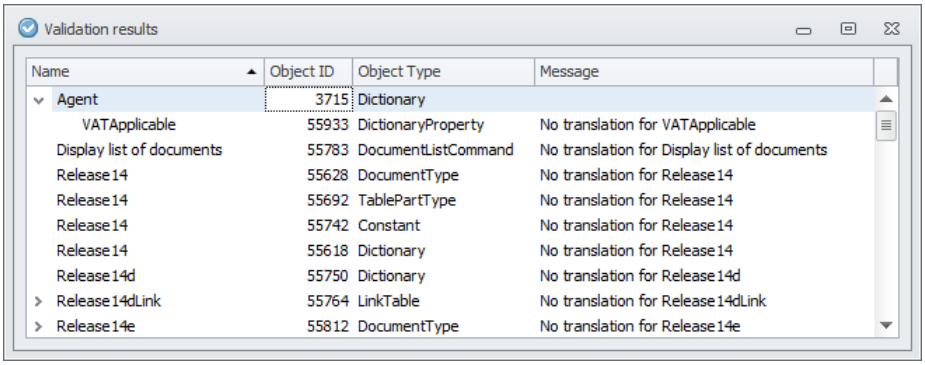
4. When the shortcut menu command selecting or by clicking + name for a new metadata branch is requested:



Process can take several minutes if the considerable history is saved up. After creation of the branch start an application server on a new branch.

Metadata error check

The check of metadata of the current version on errors is carried out by clicking the key button Validate. The found errors are displayed in the form Validation results. Top-level objects, such as dictionaries, types of documents and scripts, are opened by double-click to correct validation errors at once:



Among other things, the validation tool checks the existence of translations for the names of metadata objects and strings in scripts.

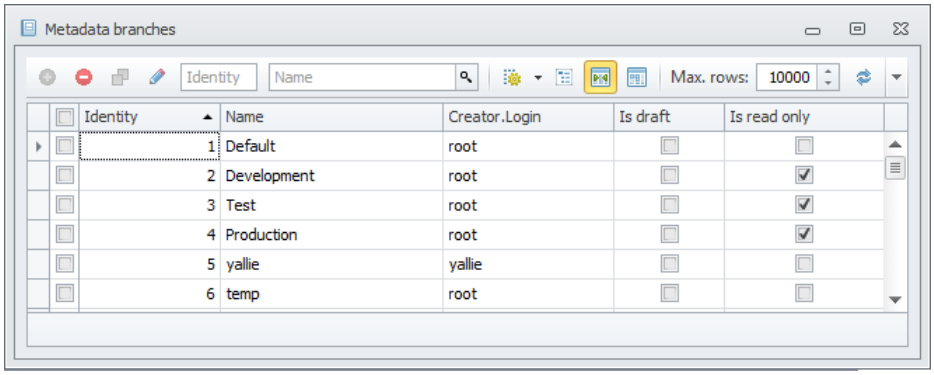
To correct errors of validation of strings in scripts it is necessary to do the following:

- if this message for the user – to replace it by resources and to translate into all languages of the system (by default it is two languages – Russian and English);
- if it is SQL inquiry – it has to begin with operators in an upper case of SELECT, DELETE, UPDATE, INSERT;
- if the situation does not fall under any of the above - to mark the string by the symbol @, in this case it will be passed by the validator without errors:

```
string someSql = @"delete from TABLENAME";
```

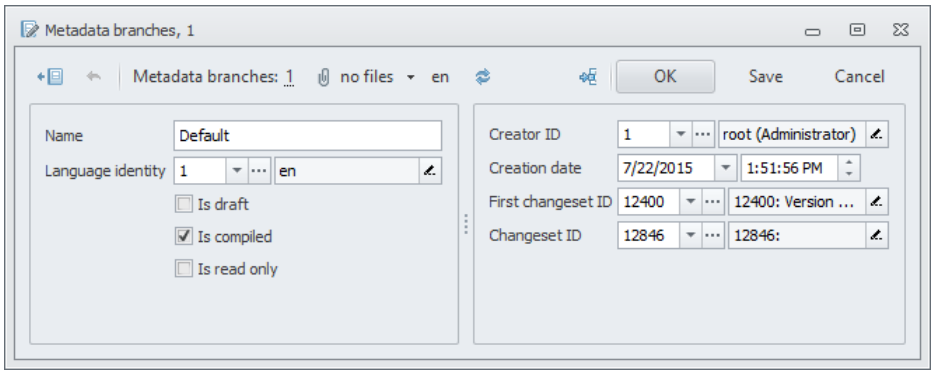
Versions tags

It is possible to view the metadata branches in the dictionary Branches:



Dictionary records can be filtered according to the *Name* of the branch (*Name*).

The tags have the following properties:

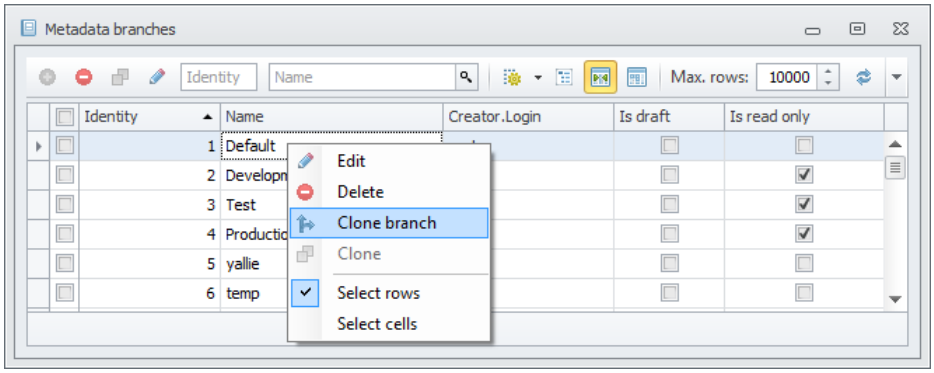


- *Name* – branch name;
- *Language* — metadata main language;
- *Creator* — branch creator;
- *Creation date* — creation date of a branch (service field);
- *First changeset* — first commit from which the branch begins (service field) ;
- *Changeset* — the last commit with which the branch comes to an end (service field);
- *Is draft* — a flag, indicating that the branch is a draft (changes from it can not be pushed to the branch Default);
- *Is compiled* — a flag, indicating, that the metadata binaries are relevant (service field);
- *Is read only* — a flag, indicating that the branch is available only for reading.

For the existing branches it is possible to change the name and the owner, and status flags as well. Service fields are not recommended to be touched.

Creation of new metadata branches

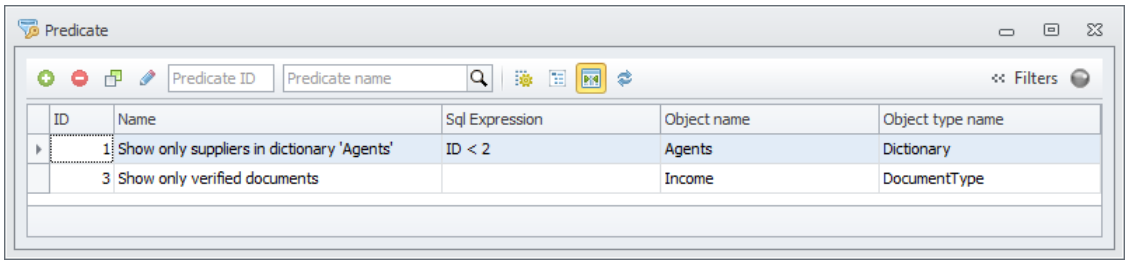
It is possible to create new metadata branches only by cloning of the existing branches (it will be offered to choose a branch for cloning by the New key button). Among versioned data there are service records (types of scripts, system dictionaries, etc.), platforms required for the correct operation, so the creation of completely empty branches are not supported. When cloning the branches there is a copying of all versioned data from the original branch, so the cloning can take a long time:



In the branches dictionary the cloning always makes a copy of the last (current) metadata commit revision on the selected branch. If the metadata on an original branch are not fixed, the cloning is impossible. If for cloning it is necessary to choose older revision of metadata, it is necessary to use a form of history of versions changes of metadata.

Predicates

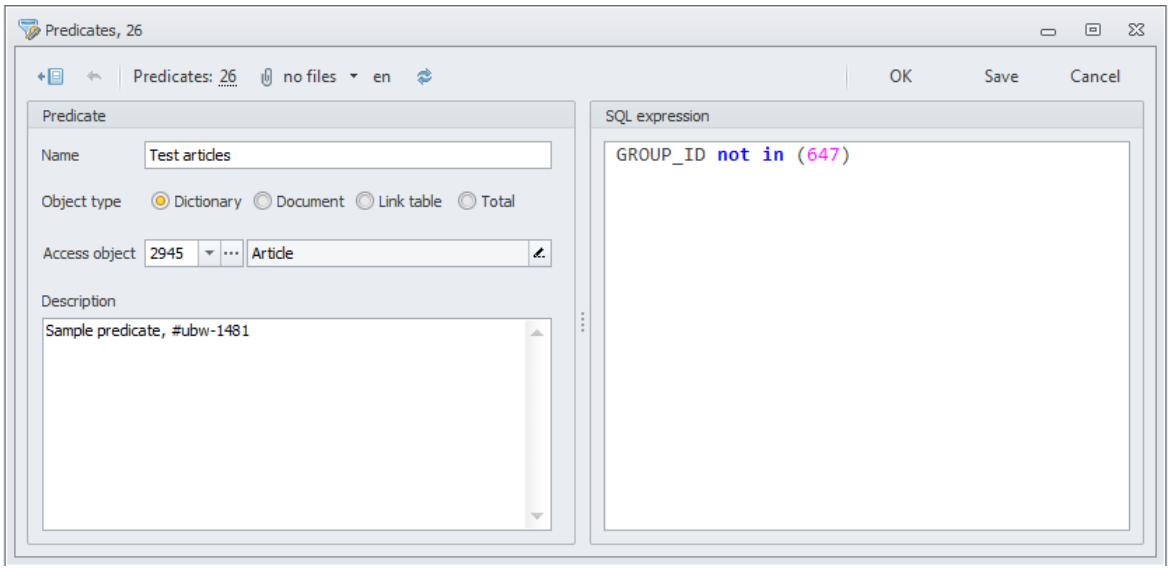
Predicates are applied to object for the purpose to restrict access to it. The list of all predicates can be found in the dictionary Predicate:



Predicative access is supported for the following objects of metadata:

- dictionaries;
- link tables;
- documents;
- totals.

The predicate has the following properties:



- *Name* – predicate name;
- *Access object* – object type is Dictionary, Document or Total (*Dictionary*, *Document* and *Total* respectively) and directly object to which the predicate is applied;
- *SQL expression* – expression in the SQL language;
- *Description* – description of the predicate action.

Fast access tools

The tools of the group are used for fast access to metadata objects:

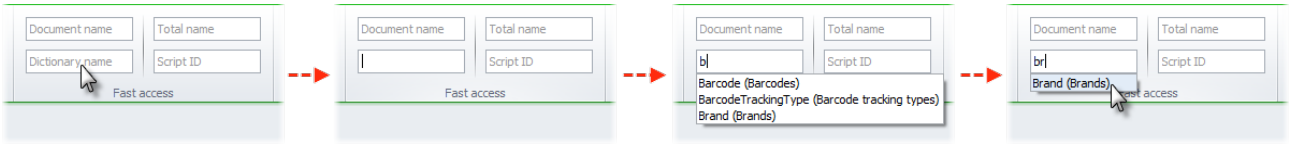
- *Document name* – selection of the document type by its name (*Name*) and description in user's language (*Caption*);
- *Dictionary name* – selection of the dictionary by its name (*Name*) and description in user's language (*Caption*). Selection is carried out only among non-system dictionaries;
- *Total name* – selection of the total by its name (*Name*) and description in user's language (*Caption*);
- *Script ID* – selection of the script by its ID (*ID*) and the class name (*Name*).

The selected metadata object opens in the edit form of corresponding dictionary.

When working with fast access tools, the options of code completion technology IntelliSense are available.

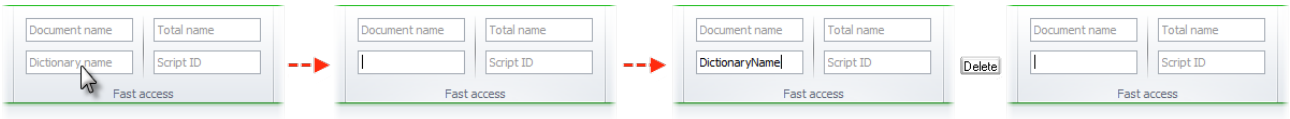
IntelliSense

When the text is entered in the control elements supporting *IntelliSense*, the system offers a choice from the list of objects, which name includes the entered fragment:

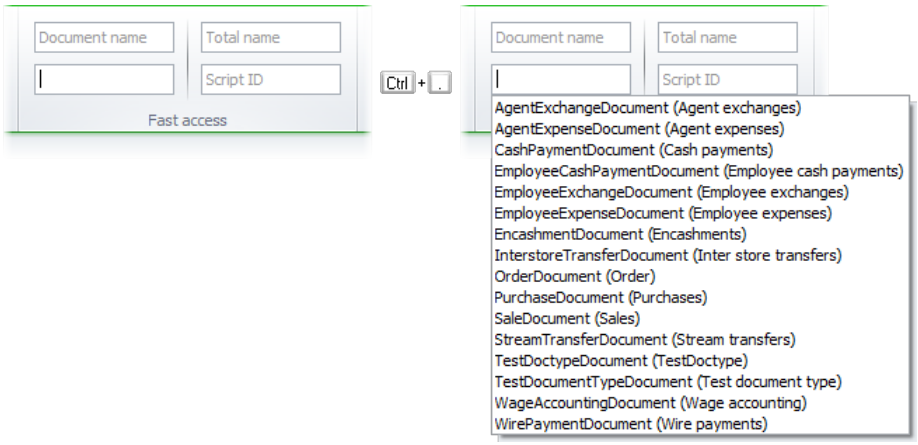


Selection is carried out with a click of the left mouse button on the object in the code completion list or by pressing the key Enter (in that case, the entered text must coincide fully with the object name in the list, even if this is the only object in the list).

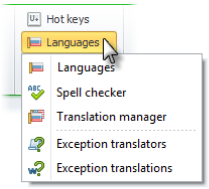
The entered text can be deleted by pressing the key Delete:

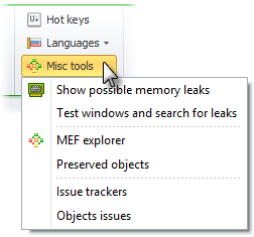


All possible options of the values, which can be entered in the control element, can be viewed too by pressing shortcut keys Ctrl + .:



Ribbon Misc



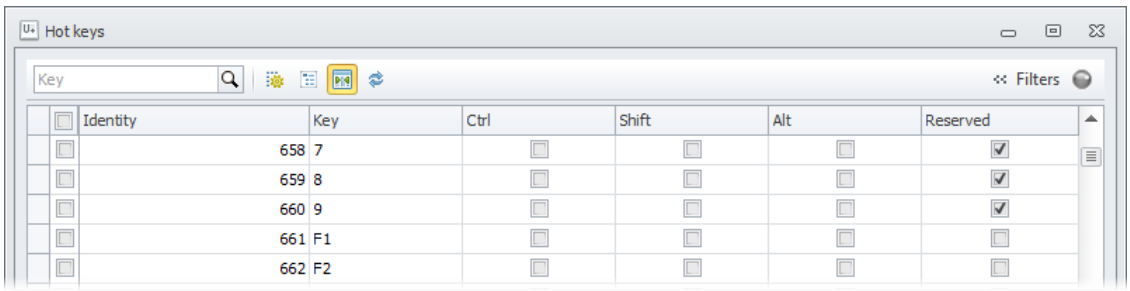


The group of Tools includes the following:

- *Hot keys* – a dictionary of all the hot keys used in the system;
- *Languages* — localization Tools menu:
  - *Languages* – dictionary of the system languages;
  - *Spell checker* – by means of which the spell check is realized in the system;
  - *Translation manager* — the tool for editing of the metadata translations;
  - *Exception translators* – dictionary of the user translators of exceptions – simple services who take exception to the input and return the translated result. The user translators are carried out after system translators, but before exception translations (*Exception translations*), from which they differ, use of scripts, that gives more flexible approach to processing of exceptions, but does them a little more difficult in application;
  - *Exception translations* – a simple tool, that allows presenting system exceptions (errors) in accessible language for an ordinary user.
- *Misc tools* — other tools for debugging:
  - *Show possible memory leaks* — report of the memory leak detector;
  - *Test windows and search for leaks* — a command, in turn, opening the available forms of the client application and showing the report of the memory leak detector;
  - *MEF Explorer* — a tool of debugging of MEF-container of the application;
  - *Preserved objects* — the list of the preserved objects, that do not need to be deleted at the comprehensive cleaning of the base (infrastructure dictionaries that do not relate to the application area).
  - *Issue trackers* — bug-trackers dictionary;
  - *Object issues* — a tool for viewing of applications, according to which the objects of metadata were changed.

Hot keys

The list of all shortcut keys available for use in Ultimate AEGIS® system can be found in the dictionary Hot keys:



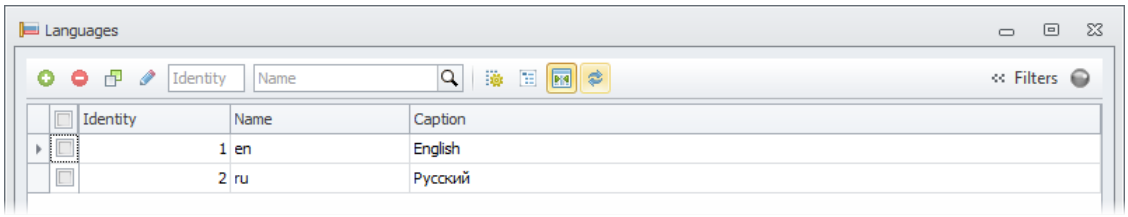
The list can be filtered by non-functional *key* (*Key*) used in the shortcut.

The shortcuts already in use are marked with the flag *Reserved*.

Tag search

Languages

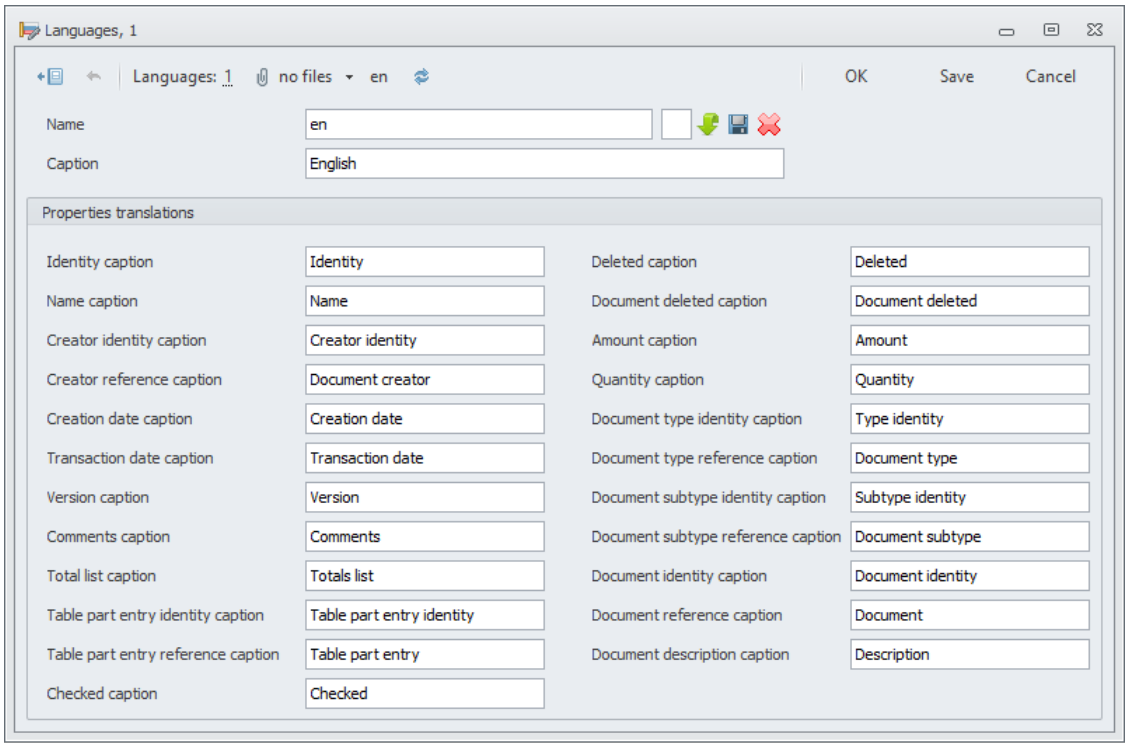
The list of all system languages Ultimate AEGIS® can be found in the Languages dictionary:



The dictionary records can be filtered by the language *Name*(*Name*).

Language with the identifier 1 is a system language by default.

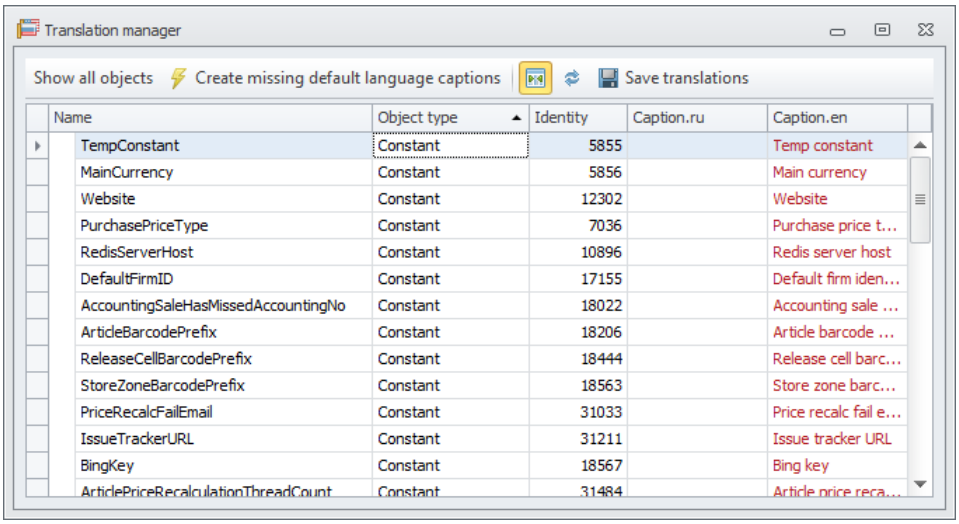
The language has the following properties:



- *Name* – name of the language;
- *Caption* – description of the language;
- *Icon* – language icon (with the size of 16 x 16 pixels).
  The buttons to the right of icon preview area allow:
  – loading the icon;
  – saving the icon previously downloaded to the computer;
  – deleting the icon;
- *Properties translations* – set of the translations of system properties.

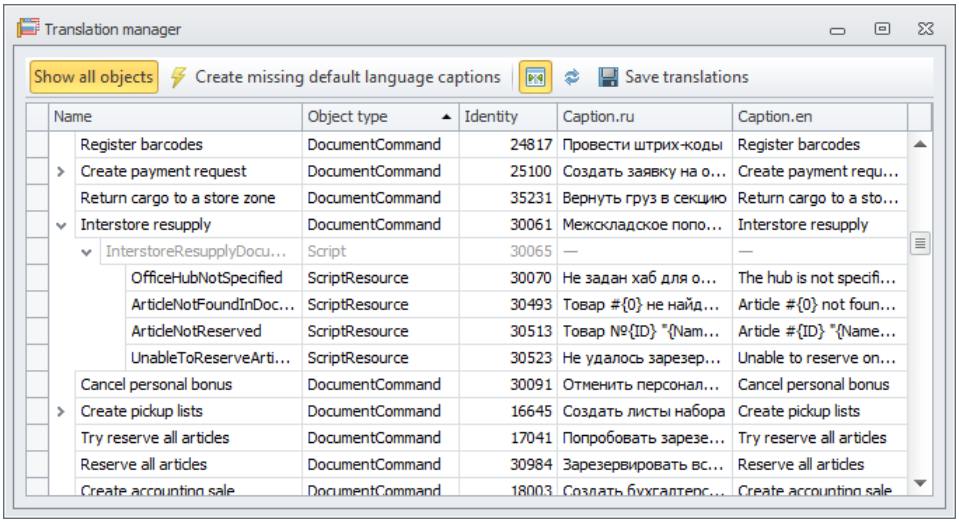
Translation manager

Translation Manager is a tool for centralized editing of the translations of all metadata objects. It represents a tree of objects with the edited columns “Caption” corresponding to all languages, supported in system. At start the translation manager automatically shows all objects for which there are missing translations on the current branch of metadata:



For objects which have no translation even on language by default, the text in the corresponding column is highlighted in red. To create translations automatically in the language by default , it is possible to use the command «Create missing default language captions». At the same time as the translations the values will be taken from the column Name, divided into separate words (for example, IssueTracker → Issue tracker).

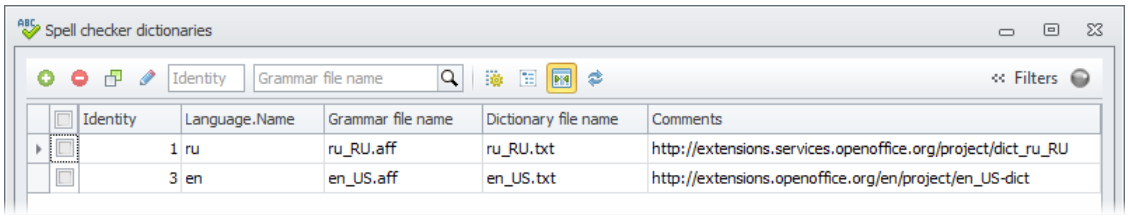
If all objects already have translations into all languages, the translation manager's form at start will show an empty list: everything is all right. To display all the metadata objects and their translations it is possible to use the button-switch «Show all objects»:



Objects, that do not require translations are displayed in pale-gray in the list, and in the Caption column the dash is shown for them. Double-click on the object line opens the object for editing (it is supported only for top-level objects, such as dictionaries or scripts: to open a property of the directory or, for example, the resource of the script for editing is possible only in a part of the object-parent). "Save translations" button keeps the made changes. When closing the translation manager checks if there was changes from the moment of the last save and asks again if it is necessary.

Spell checker

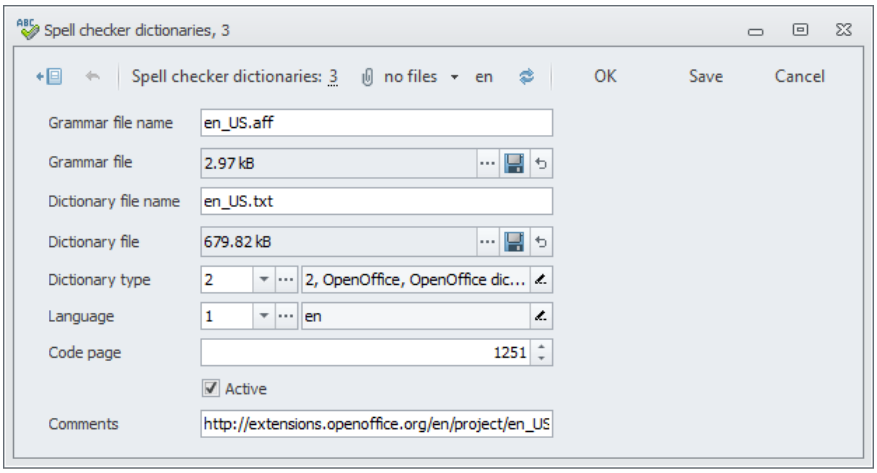
In the system Ultimate AEGIS® speell cheecking is realised, which is carried out by means of dictionaries. The words can be found in the Spell checker dictionaries:



Dictionary records can be filtered by the *Name of the file of the grammatical dictionary* (*Grammar file name*).

It is not strongly recommended to start more than one active (present) dictionary for one language.

The dictionary has the following properties:



- *Grammar file name* – grammar file name containing rules of word formation (affixes – prefixes, suffixes, endings, etc.). The words (roots) are contained in the file *Dictionary file*;
- *Grammar file* – file of grammar dictionary. If the dictionary is chosen, its size is displayed at the control element.
  Control key buttons allow:
  - ▪ – by clicking the key button the dialog of dictionary loading is opened (instead of present one);
  - ▪ – by clicking the key button it is possible to save the dictionary, loaded earlier from the database to the local disk of the computer or other available media;
  - ▪ - by clicking the key button the dictionary is opened in the program, associated with the operating system of its type file;
- *Dictionary file name* – dictionary file name, containing directly words (roots);
- *Dictionary file* – dictionary file;
- *Dictionary type* – type of used dictionaries:
  - ▪ *ISpell* – dictionaries of a format ISpell;
  - ▪ *OpenOffice* – dictionaries of a format OpenOffice;
- *Language* – language, which check is carried out by means of these dictionaries;
- *Code page* – code of a used code page;
- *Active* – the flag, indicating the need to use a dictionary. The dictionary with the removed flag *Active* is not used;
- *Comments* – comments to the dictionary.

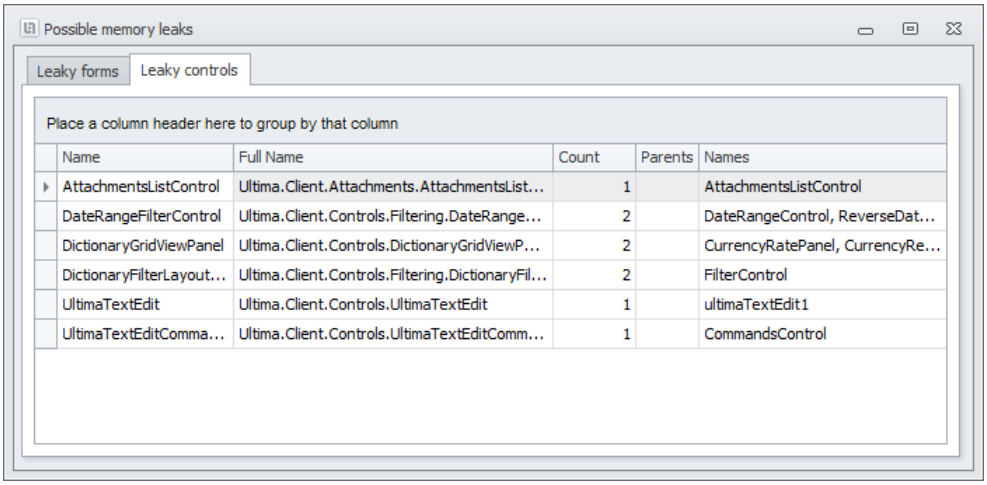
Exception translators

Translators exceptions are services that take exception to the input and return the translated result, suitable for displaying to the end user. Translators distinguishes from exception translations by the use of scripts, which gives a more flexible approach to exception handling, but makes them more difficult to use. For more information about exceptions translators see section Scripts.

Memory leak detector

Memory leak detector represents background service of the client application which traces the forms and elements of the control, remaining in the memory after calling of their method Dispose. Certainly, not all memory leaks have such nature, however it is the most widespread, resource-intensive and at the same time simple leaks for search.

A command *Show possible memory leaks* allows requesting the current report of the leaks detector at any time. The report represents a table with two tabs: a list of potential leaks of forms and list of leaks of controls elements. As a rule, forms and elements represent a connected graph in which the leak of some one element keeps in memory the entire graph.



Name	Full Name	Count	Parents	Names
AttachmentsListControl	Ultima.Client.Attachments.AttachmentsList...	1		AttachmentsListControl
DateRangeFilterControl	Ultima.Client.Controls.Filtering.DateRange...	2		DateRangeControl, ReverseDat...
DictionaryGridViewPanel	Ultima.Client.Controls.DictionaryGridViewP...	2		CurrencyRatePanel, CurrencyRe...
DictionaryFilterLayout...	Ultima.Client.Controls.Filtering.DictionaryFil...	2		FilterControl
UltimaTextEdit	Ultima.Client.Controls.UltimaTextEdit	1		ultimaTextEdit1
UltimaTextEditComma...	Ultima.Client.Controls.UltimaTextEditComm...	1		CommandsControl

The detector report does not allow eliminating the leaks, it only states the existence of a potential problem. For search and elimination of the concrete reasons of leaks you should use a specialized memory profiler.

A command *Test windows and search for leaks* automatically opens and closes all forms of dictionaries and documents, available on the client. This process can take several minutes and can display errors if some of client forms do not work. When all the forms are closed, the command will display the leak detector report or a message that the potential leaks are not detected. The similar tool allows quickly estimate the existence of problems in the project, for example, before a deployment of the next application version.

The message, that leaks are not found, is not a guarantee that there are no memory leaks in the program! The built-in detector has limited functionality and it is not a replacement of the real memory profiler at all. To guarantee the lack of leaks, you should analyze the snapshots, received by means the profiler. However, if the memory leak at the level of a form or elements control exists, the built-in detector, as a rule, will be able properly to outline the circle of suspects.

MEF Explorer – debugging

 The MEF platform (eng/rus) is the framework for the Ultimate AEGIS® system. It assembles an application from independent components, such as kernel services, scripts and applied classes. MEF is used both in server and client parts, making the program to have an identical modular structure in both cases and use single API, which is well documented and rather popular.

MEF is a late binding system based on the comparison rules. The comparison is carried out between so called imports and exports provided by the components. To get a component ready to go, all its imports must be compared with exports of other components. Binding is carried out during the execution of the program, which ensures the desired flexibility. The components to build the program may be developed simultaneously by separate programmers' teams: e. g., the form of a client application may use a server service, which is under construction, provided that the service interface is formally coded.

A downside of such flexibility is binding errors. If in the process of execution the application needs a service that is not yet implemented, a binding error will occur. The program components may have sophisticated relations, and even if one such relation is not found, the whole component becomes useless.

Unfortunately, the diagnosis of such bugs is difficult. All that MEF knows at the moment of occurrence of a binding error is that one of the binding rules cannot be complied with. A typical text of the bug in such case is as follows:

```
The composition produced a single composition error. The root cause is provided below.
Review the CompositionException. Errors property for more detailed information.

1) No exports were found that match the constraint:
ContractName          Ultima.Scripting.IUserCommand
RequiredTypeIdentity  Ultima.Scripting.IUserCommand
RequiredMetadata
    ScriptID          (Ultima.Scripting.IScriptMetadata)

Resulting in: Cannot set import '
    ContractName            Ultima.Scripting.IUserCommand
    RequiredTypeIdentity    Ultima.Scripting.IUserCommand
    RequiredMetadata
        ScriptID            (Ultima.Scripting.IScriptMetadata)' on part '(name)'

Element:
    ContractName            Ultima.Scripting.IUserCommand
    RequiredTypeIdentity    Ultima.Scripting.IUserCommand
    RequiredMetadata
        ScriptID            (Ultima.Scripting.IScriptMetadata) --> Unknown Origin

   at System.ComponentModel.Composition.CompositionResult.ThrowOnErrors(AtomicComposition
atomicComposition)
   at
System.ComponentModel.Composition.Hosting.ImportEngine.SatisfyImportsOnce(ComposablePart
part)
   at
System.ComponentModel.Composition.Hosting.CompositionContainer.SatisfyImportsOnce(Composab
lePart part)
```

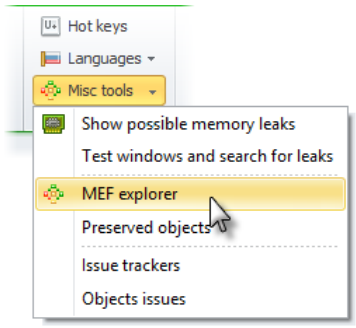
It is only clear from the text that some component cannot be provided upon the system request. Which component exactly is the cause of the error and how to correct it is unclear.

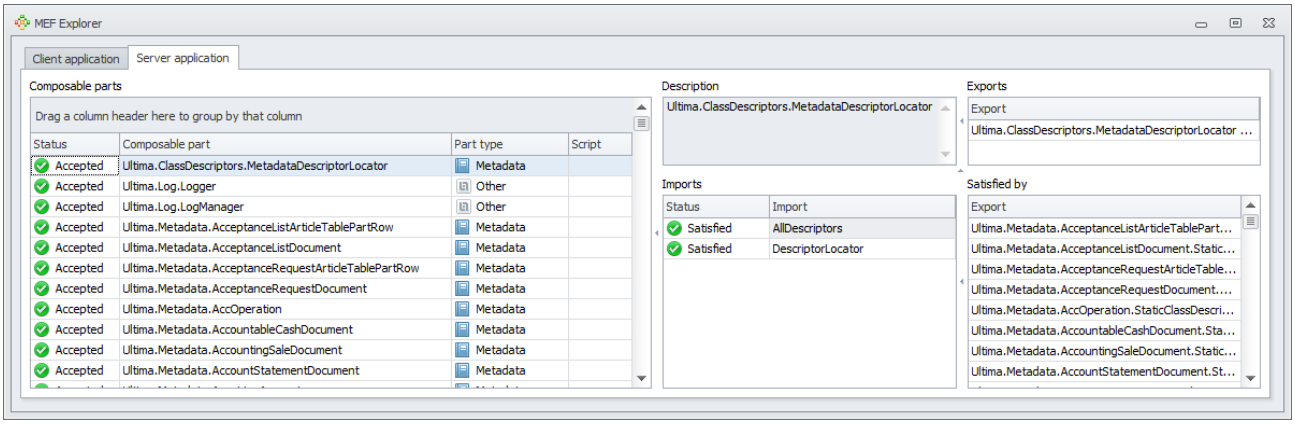
Since it is only applied errors of such kind that are of interest to programmers, the range of possible causes narrows down to two typical situations:

- error in a script (not necessarily in that one requested);

- error in a client form (or in one of its relations).

To investigate such errors on the basis of the console utility Mefx, a tool called *MEF explorer* was developed. It shows the structure of MEF catalogs for client and server parts (in the corresponding tabs).





If no errors occurred, all component imports are compared with exports of other components, and all the composable parts are supplied with the status *Accepted*.

Of course, other statuses are of greater interest. Any binding error means that the component was rejected due to one of the following reasons:

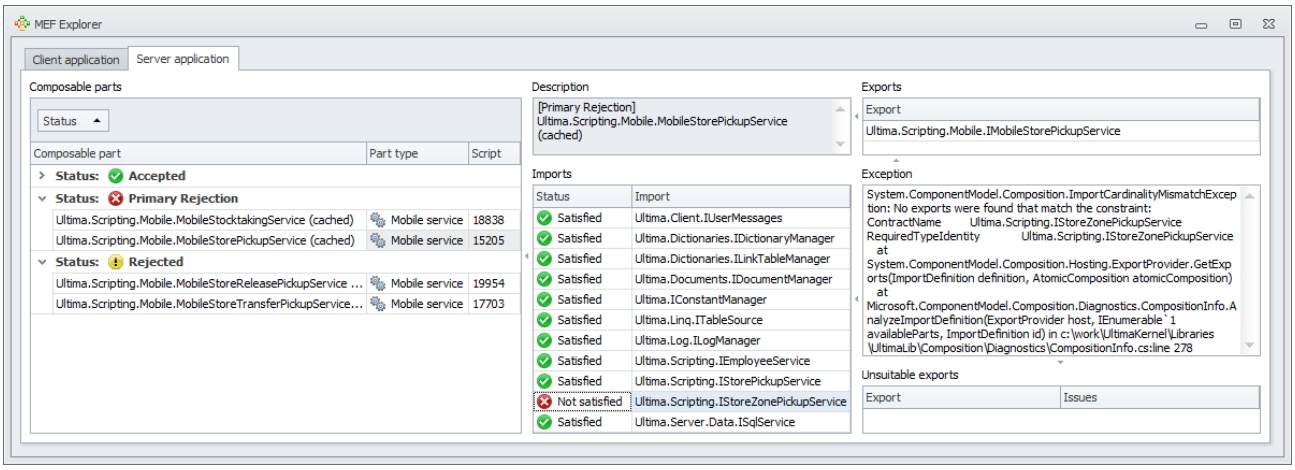
Primary Rejection – particular component's imports lack corresponding exports;

Rejected – component's exports exist, but cannot be created, as their imports, in their turn, lack corresponding exports.

Let's see, what actions the application programmer will perform, once the error occurred.

Suppose that the binding error (No exports were found that match the constraint...) occurs during the *MobileStorePickupService* request. However, the real problem is never in the request. If to open the script in an editing program, we will see that it is compiled without errors. To identify the reason, run *MEF Explorer* and search for the component that caused the error in the *Composable parts* list by its statuses, *Rejected* or *Primary Rejection*, and name.

When selecting this component in the *Imports* list, to the right of the component, there will be shown all imports that are either bound or unbound with exports. Viewing unbound imports, we can find out the real cause of the error:

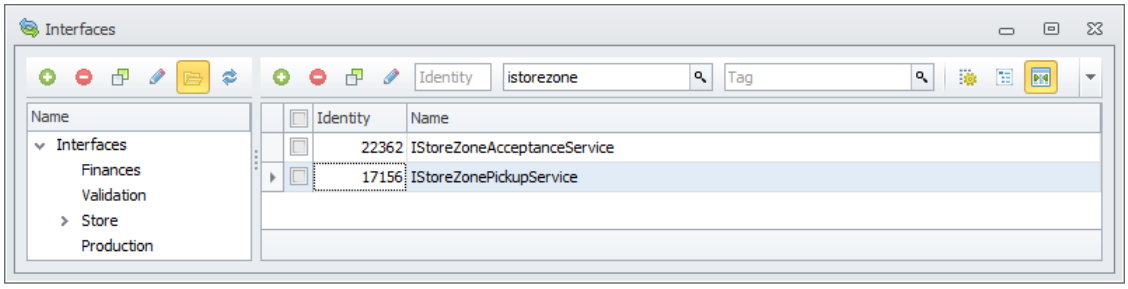


In the example above, we can see that the problem is in *IStoreZonePickupService*. This application service is absent in the server-side catalog. There may be the following reasons for that:

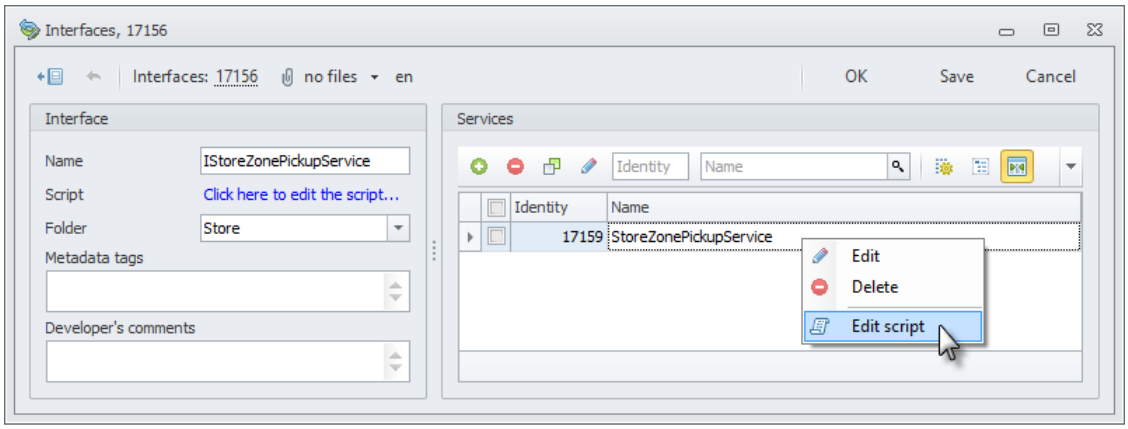
- the service has an interface, but it is not implemented so far;
- the interface is implemented, but there are compilation errors;
- the MEF Cache of the service script is empty for some reason (probably, as a result of merging of metadata versions).

To correct the error, you need to:

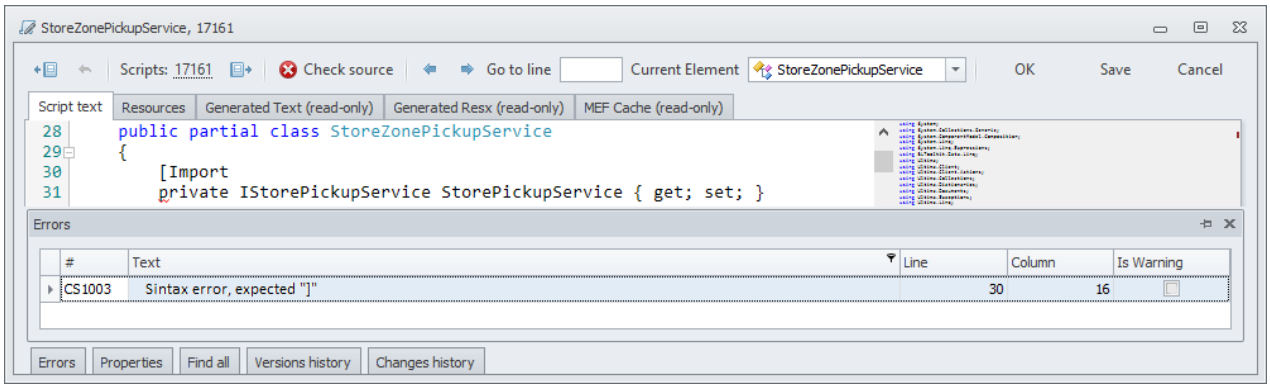
- find the *IStoreZonePickupService* interface in the *Interfaces* dictionary:



- check if the interface is implemented (if not, add implementation):

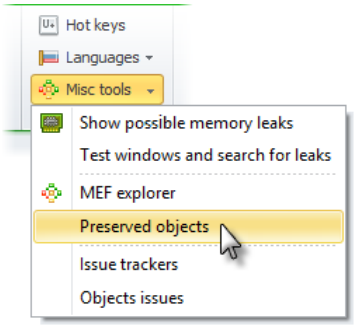


- if implemented, repair the implementation script:

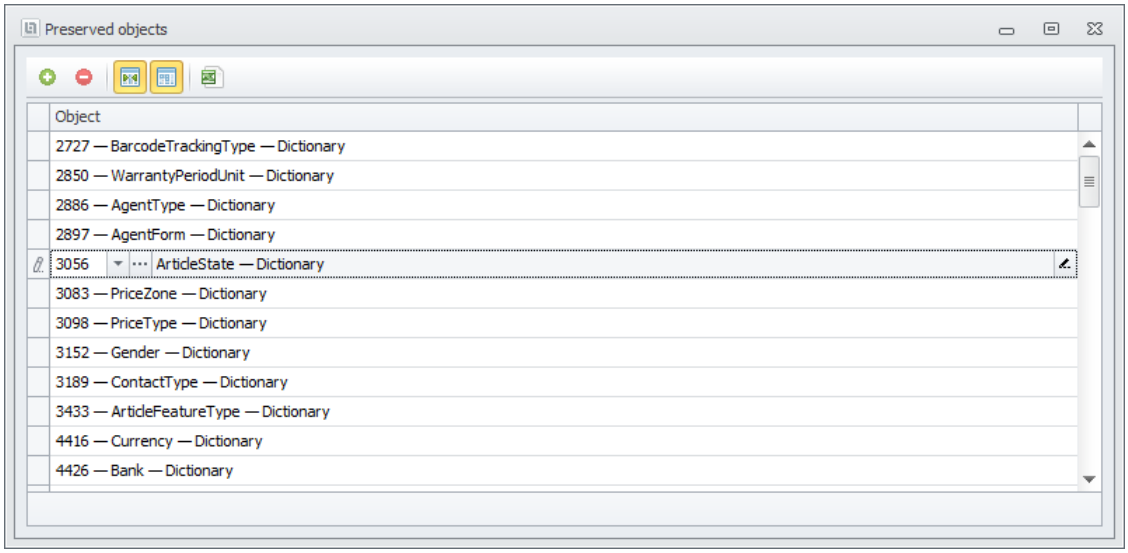


Preserved objects

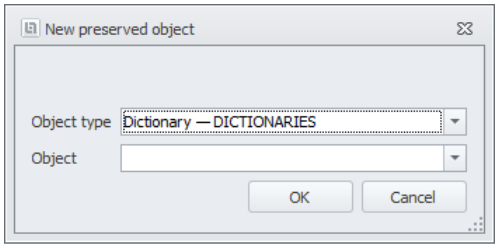
In the process of development there can be a need to clear applied base of the data, created as a result of testing for example. But at the same time there can be such dictionaries and link tables which data can not be deleted.



The list of such objects is stored in the list of the preserved objects *Preserved objects*. This list is the declarative one, that is applied developer who is carrying out the cleaning of the database is guided by it when making requests to delete the data and does not include the list of the mentioned objects in them:

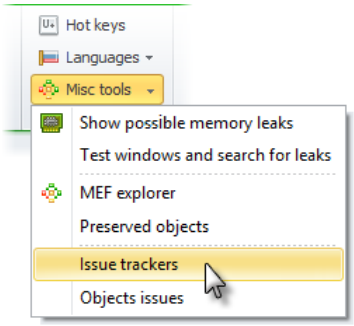


The object can be added to the list can be deleted by the corresponding key buttons in the toolbar. When adding a new object it is previously necessary to choose its *Type* (*Object type*).



Object issues

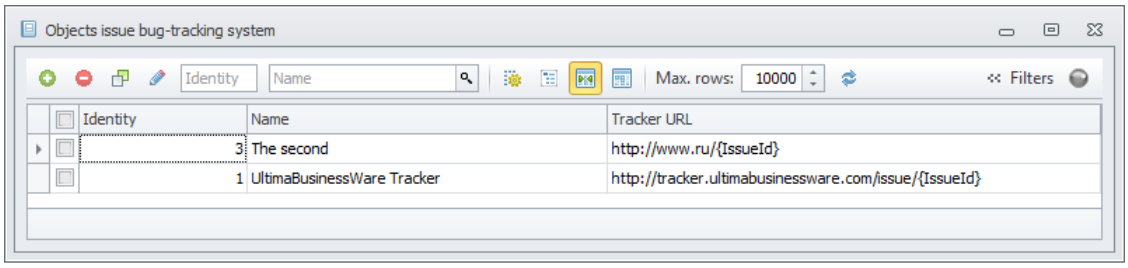
In the system Ultimate AEGIS® issue & bug tracker is realized (further *tracker*).



When changes fixing an application developer can specify the request identifier in tracker, within which changes were made to metadata. In case identifier input all objects which changes are fixed will be marked as changed within this request. The entered identifier shall match the request identifier in URL tracker – in this case the request can be opened in a tracker from metadata object.

At the same time it place restrictions on the application developer. If changes made within two and more requests are fixed they will not be marked correctly with IDs. In case of operation with several tracker requests it is necessary before operation within the following request to record changes by previous.

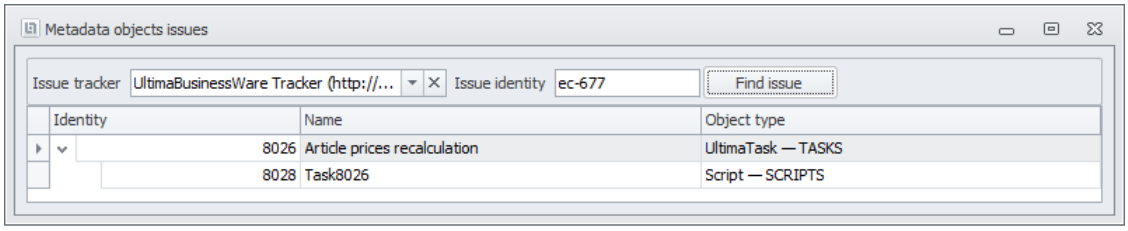
Use of several trackers, which list can be found in the dictionary *Objects issue and bug-tracking system,* is possible (opens the *Issue trackers* command):



Name (*Name*) and URL (*Tracker URL*) should be set for a tracker. URL needs to be set in the ***http://tracker.ultimabusinessware.com/issue/****{IssueId}* format (the part which needs to be changed to appropriate URL is highlighted **in bold).** *{IssueId}* part should be left invariable in the resultant address, it is replaced with number of the specific request.

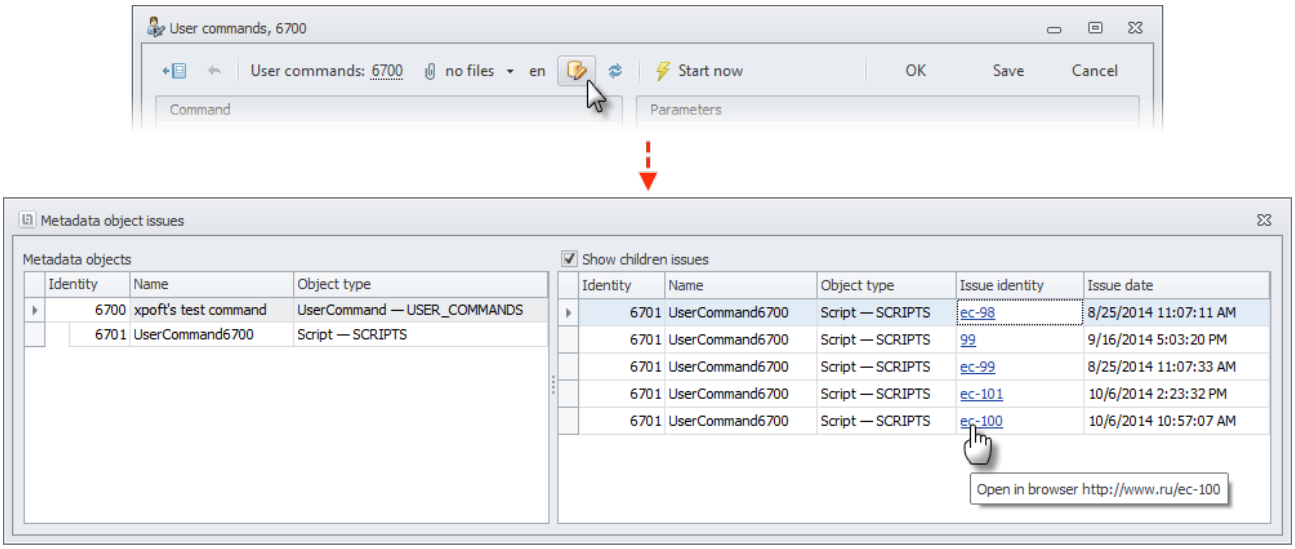
It is possible to find all changes entered to all metadata objects within the specific request with *Objects issues* tool.

In *Metadata objects issues* form it is necessary to select a tracker (*Issue tracker*), to enter a code of its request (*Issue identity*) and press "Find issue". All object list when their changes fixing this identifier was entered will be output in search results:



- *Identity* – a code of the changed metadata object;
- *Name* – object name;
- *Object type* – system object time.

It is also possible to look within what requests the specific metadata object changed. For this purpose in the form of its editing it is necessary to click toolbar button. The opening form *Metadata objects issues* is divided into two parts:



In the left part of the form *Metadata objects* metadata objects are displayed in a tree structure. The first level is an object from which editing form this form was opened and also all its child objects which changed within any requests:

- *Identity* – a code of the changed metadata object;
- *Name* – object name;
- *Object type* – system object time.

In the right part of the form the list of requests within which the object selected at the left changed is displayed. Flag activation *Show children issues* over the list it is possible to output in it also requests within which its child objects changed:

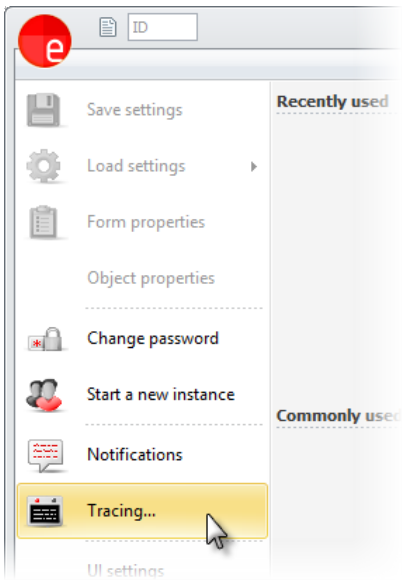
- *Identity* – a code of the changed metadata object;
- *Name* – object name;
- *Object type* – system object time.
- *Issue identity* – tracker request code. by clicking the link the request opens directly in a tracker;
- *Issue date* – change date according to the request.

Tracing

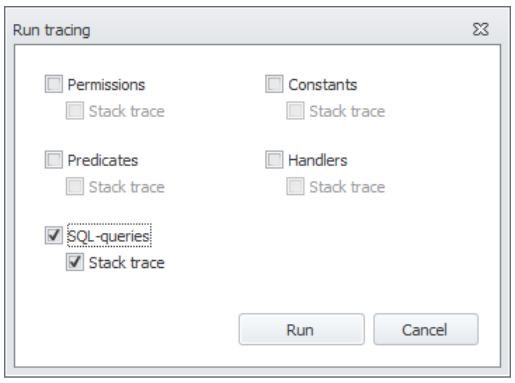
Tracing is used to trace the application execution. In tracing mode it is possible to track the sequence of the commands and values of query parameters that makes it easier to detect the errors.

Tracing tool is included in the basic module, since the use of its functionality may be required in the session of the ordinal user who isn't granted the developer permissions.

Tracing is started using the *Tracing...* menu item.



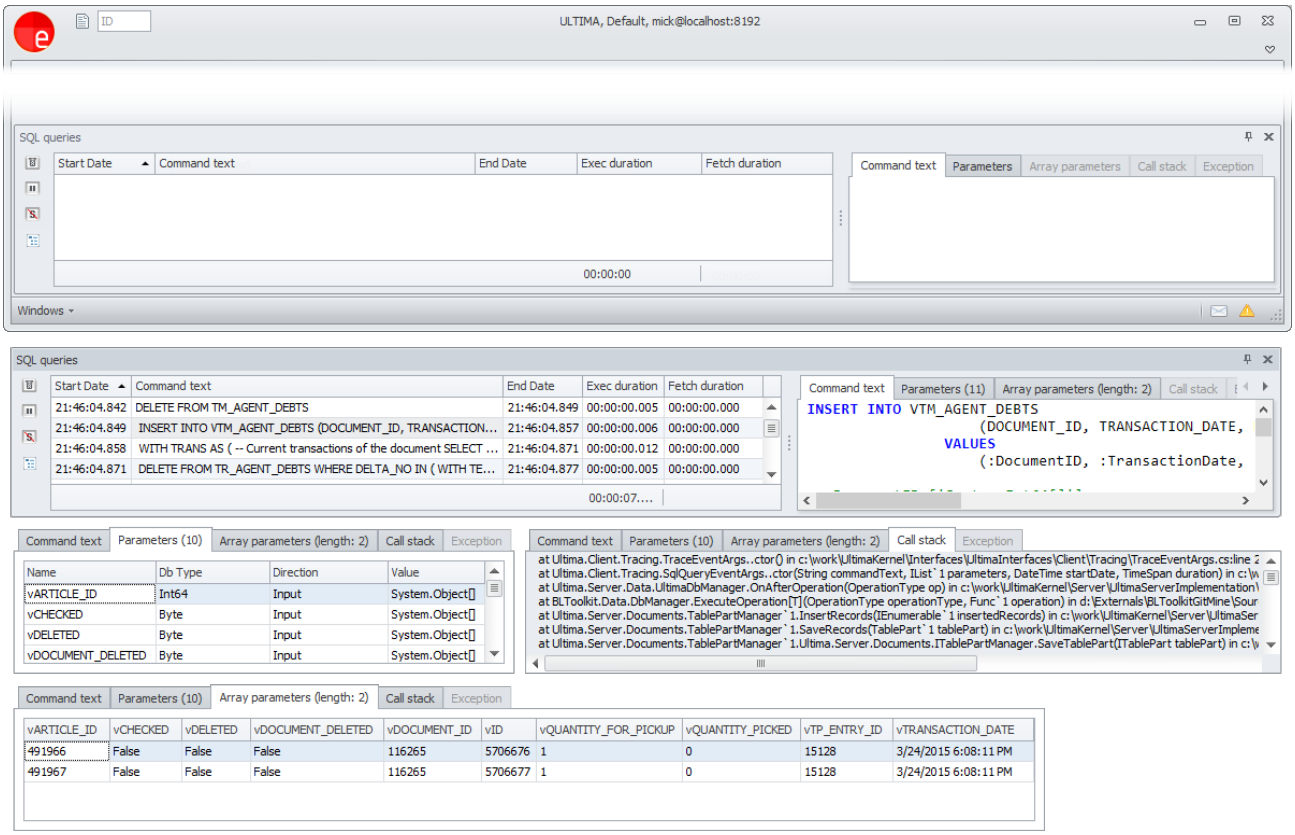
In the opened window *Run tracing* it is possible to choose objects which work needs to be monitored:



SQL queries

In the opened window of *SQL queries* the following functionality is available:

- click the key button to erase tracing results, clearing the window;
- click the key button to stop tracing or to start it again;
- click the key button to turn off the stack trace function or turn it on;
- click the key button to open and hide a group panel:



Columns in the list of SQL queries:

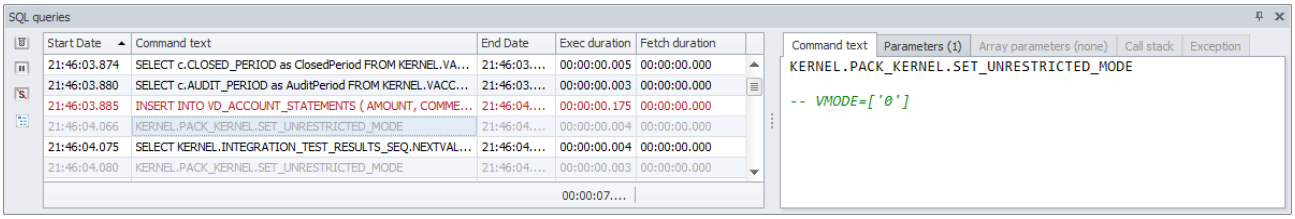
- *Start Date* – start date and time of the query execution;
- *Command text* – SQL query text;
- *End Date* – end time of the query execution;

- *Exec duration* – time of the query execution at the database server;
- *Fetch duration* – data transmission time from the database server to the application server.

In the tabs to the right of the requests list the following items are located:

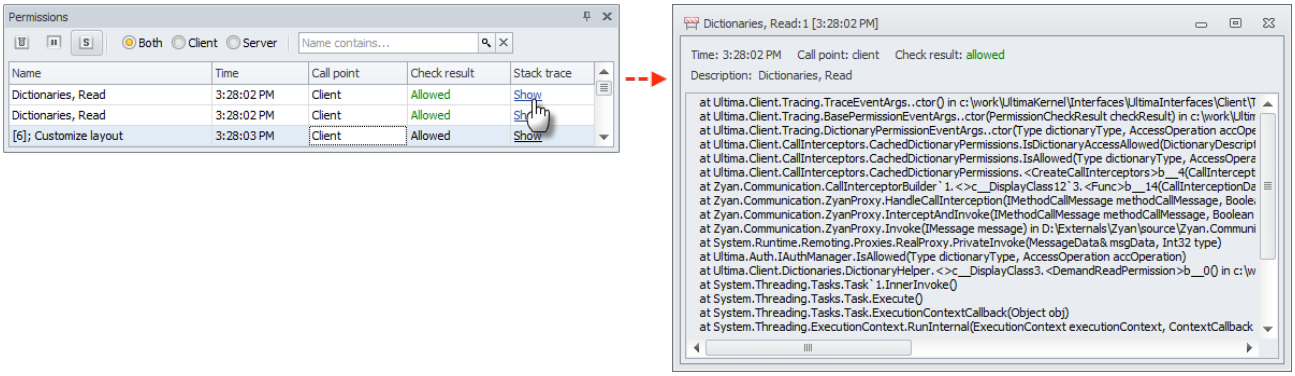
- *Command text* – SQL query text;
- *Parameters* – query parameters;
- *Array parameters* – parameters arrays (in this case, as in the shown example, in the tab *Parameters* they are not displayed);
- *Call stack* – call stack;
- *Exception* – exception text, if it has arisen when performing the query.

Requests throwing exceptions are highlighted in red in the list. Infrastructure queries not related to business logic are highlighted in gray:



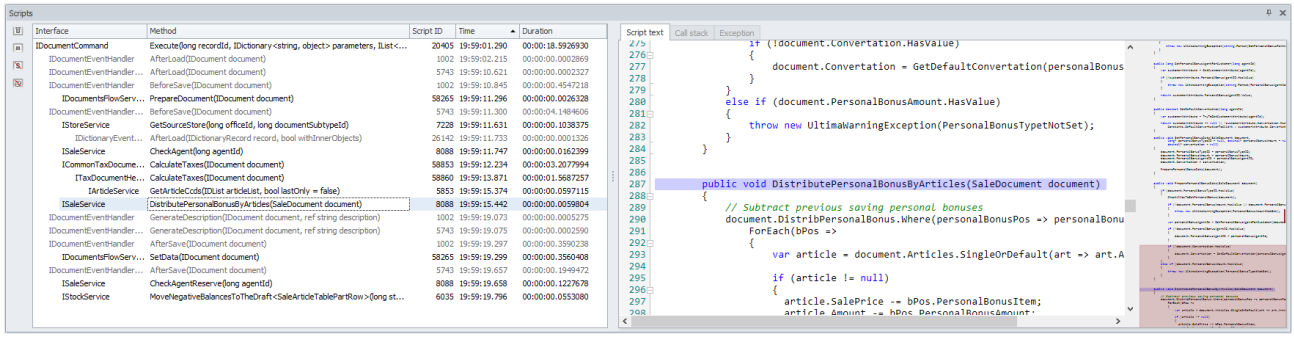
Permissions

It is possible to filter results of permissions tracing event by their source (on the server or the client) or by the name:



Scripts

Script tracing displays the script call hierarchy during the remote server calls (such as document commands). Calls are arranged in a tree-like structure according to their execution order. Script tracing detects only calls between different scripts, that is, the script calling its own different methods is represented with the single line in the tracing. When the tracing event is selected, the right panel loads and displays the corresponding script text and selects the method related to the selected tracing event:



Script tracing tree list displays the following columns:

- *Interface* – script interface name;

- *Method* – the name of the called method;
- *ScriptID* – script identity;
- *Time* – starting time of the script execution;
- *Duration* – the duration of the executed method.

Server calls that throw exceptions are highlighted in red. When selected, these events activate the Exception tab on the right panel that displays the exception details.

KERNEL scheme

The database consists of two schemes: kernel and subject ones. In the subject scheme, the tables are created and the data of subject area is stored, and the metadata describing business logic and business objects are stored in the kernel scheme.

The structure for metadata storage is described in this chapter.

Data types

In the system Ultimate AEGIS® the following data types are used:
- *int* – integers, 64-category;
- *long* – integers, 64-category;
- *decimal* – decimal, 96-category;
- *bool* – for storage of logical values;
- *string* – a line in size no more 2Kb;
- *text* – a text in size no more 2Kb;
- *LargeText* – a line of the unlimited size;
- *date* – date (without time);
- *DateTime* – date and time (it is broadcast on time zones);
- *byte[]* – array (*binary*, is used to store binary files).

Property types are not displayed in the documents journals and list forms dictionaries (and, accordingly, in their filters and forms of columns selection): *LargeText*, *text*; and *byte[]* (*binary*).

Dictionaries

Ultimate AEGIS® system provides the application developer with the mechanisms for creation and edition of the dictionaries describing business objects of the subject area.

Description of dictionaries is stored *in the kernel scheme* of database in the following tables:
- *DICTIONARIES* – attributes of the dictionaries;
- *DICT_PROPERTIES* – properties of the dictionaries;
- *PROP_TRANSLATIONS* – localized value of dictionary fields, translated into the language different from default one;
- *DICT_TOONEREFS* – relations of properties of the dictionaries with other dictionaries;
- *LINK_TABLES* – attributes of link tables;
- *LINK_PROPERTIES* – properties of link tables;
- *LINK_TOONEREFS* – relations of properties of the link tables with other dictionaries;
- *DICT_LINKTABLES* – relations of dictionaries with others through the link tables.

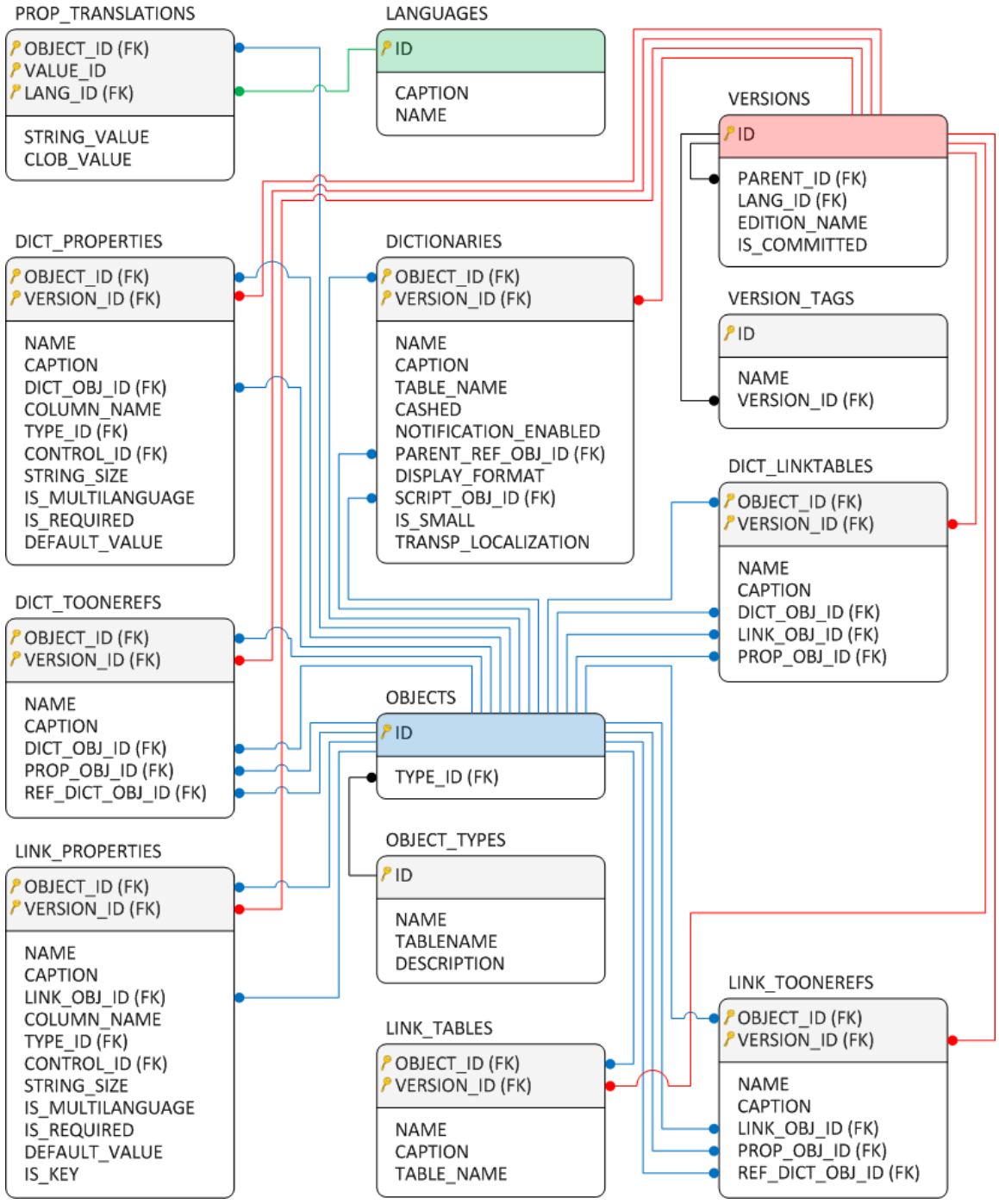
The fields of these tables will be detailed below during description of the mechanism for creation of new objects of dictionary type.

Using the tables *OBJECTS* and *VERSIONS* , configuration versioning mechanism is implemented.

Each of the tables describing the dictionaries, has two key fields related to corresponding tables:

- *OBJECT_ID* – object ID;
- *VERSION_ID* – configuration version ID.

Physical (ER) model of data looks like as follows:



The application developer can perform queries to metadata and through the views – virtual tables, obtained by retrieval from the database of all objects relating to one configuration, which ID is obtained from the current session. Therefore, while accessing the database tables and making changes to them, the developer gets access only to the configuration version selected during login to the system. For all tables of the views, prefix "V" is added to the table name.

The model of the projection of current version of data looks quite simpler in that case:

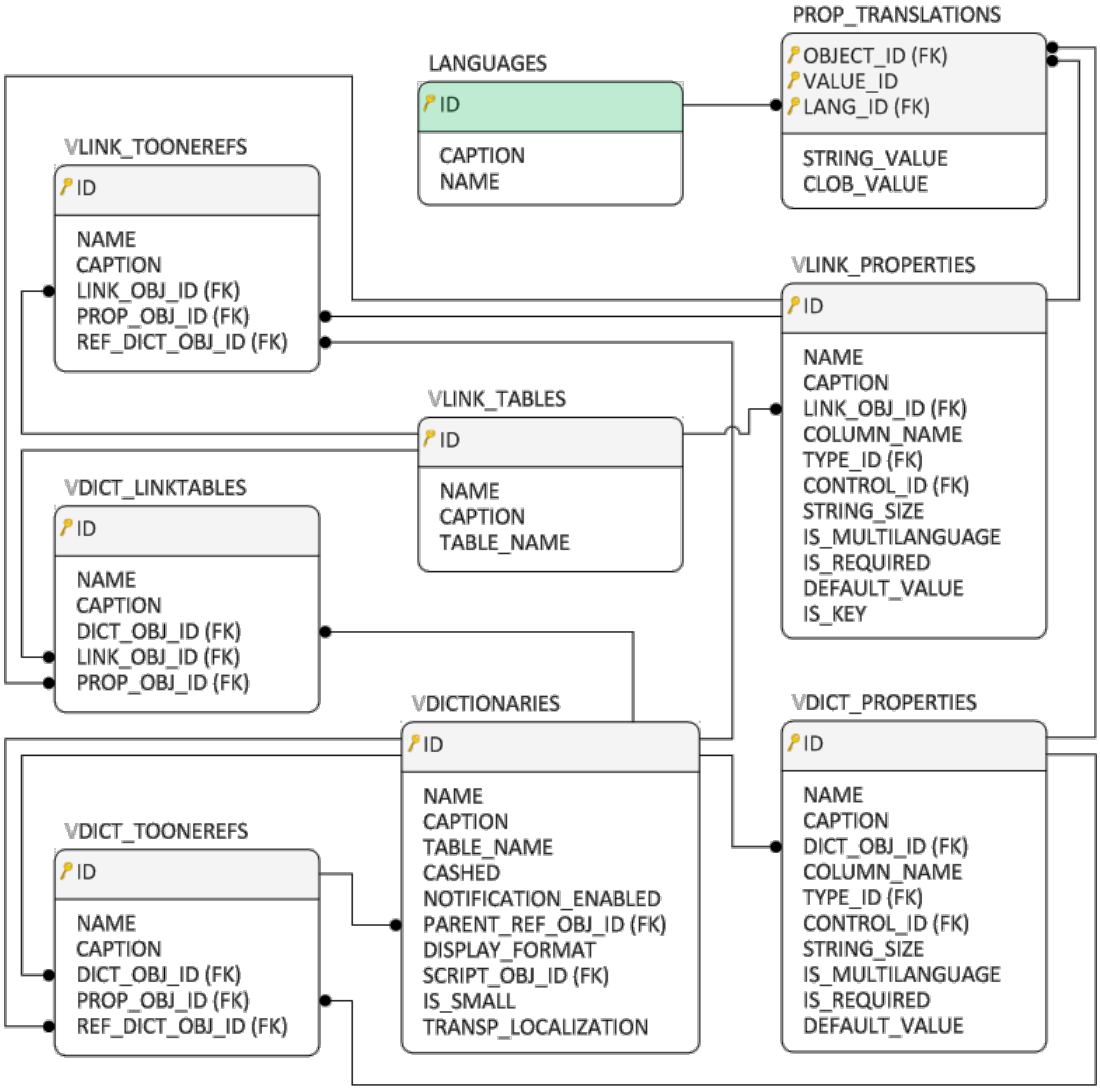


Table *PROP_TRANSLATIONS* of data model has no prefix "V", since it is not a view because it is not versioned.

Table *DICTIONARIES* keeps the attributes, which describe the dictionary and its table *in the subject scheme of* database:

- *NAME* – a name of the dictionary defines the name of generated *class*;
- *CAPTION* – dictionary name displayed in the screen forms, for example, for the dictionary of contractors Agents; the name can be "Contractors" or "Dictionary of contractors";
- *TABLE_NAME* – a table name *in the subject scheme* of database (as a result of restrictions imposed by Oracle DBMS, the name can contain only Latin letters in the upper case, figures and sign "_", moreover, the name must start with a letter and have a length of not more than 30 characters);
- *CACHED* – caching flag if being set – the dictionary is cached on the computers of end users. In case of its repeated retrieval, the data are taken from the local copy but not from database server, which can be used for the dictionaries with rarely changed data, e.g. with the list of offices.

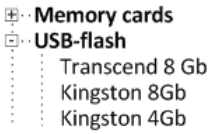
For cached dictionary, the notifications of the change are always distributed, therefore frequently changed dictionaries should not be made cached.

- *NOTIFICATION_ENABLED* – a notification flag, used for notification of the changes made to the dictionary data. For cashed dictionary, the notifications are distributed always irrespective of flag status;

- *PARENT_REF_OBJ_ID (FK)* – the attribute is used to create tree-type dictionary. A link to the table record *DICT_TOONEREFS* is used as its value, which indicates which of the properties of selected dictionary (fields of its table) is parent one (*PARENT_ID* in the example):

"Articles" dictionary

ID	NAME	PARENT_ID
1	Memory cards	NULL
2	USB-flash	NULL
3	Transcend 8 Gb	2
4	Kingston 8Gb	2
5	Kingston 4Gb	2



- *DISPLAY_FORMAT* – a format of the rows, in which dictionary records are displayed in the screen forms, when produced not in table but in the row form;
- *SCRIPT_OBJ_ID (FK)* – the handler of dictionary events (before creation, before saving, after saving, before deletion, after deletion), is created as may be necessary;
- *TRANSP_LOCALIZATION* – a flag for localization transparency (*true* – the dictionary is localized in a transparent manner, *false* – the dictionary is localized in non-transparent manner).

Localization of dictionaries is detailed in the next section.

Table *DICT_PROPERTIES* describes the dictionary properties – in fact, its table fields:

- *NAME* – name of the property, defines the name of the *property* of generated *class*;
- *CAPTION* – dictionary property name displayed in the screen forms, e.g. for the property Name of the dictionary of contractors it can be "Name of the contractor" or "Contractor's name";
- *DICT_OBJ_ID (FK)* – a link to the dictionary, the property belongs to, filled in automatically upon its addition to the dictionary;
- *COLUMN_NAME* – a name of the table field in the *subject scheme of the database* (Oracle DBMS imposes the same restrictions on the field name as the table itself);
- *TYPE_ID (FK)* – property type (see details in section Data types);
- *CONTROL_ID (FK)* – control element, which will be used by default in the screen forms for input of the values of this property, to be selected from those offered by the system. For example, for the string property, it can be string, field or string for input of password hiding the characters;
- *STRING_SIZE* limits the string length of for *TYPE_ID* property type, *string* or *text* is selected;
- *IS_MULTILANGUAGE* – a flag indicating if translation is available for that property;
- *IS_REQUIRED* – a flag indicating if the property is mandatory for fill-in;
- *DEFAULT_VALUE* – a default value, which is inserted automatically in case of creation of new element of the dictionary.

Table *PROP_TRANSLATIONS* keeps the localized values of the dictionary:

- *OBJECT_ID (FK)* – a link to localized property of the dictionary;
- *VALUE_ID* – a dictionary record in the *subject scheme* of the database, which property is being localized;
- *LANG_ID (FK)* – localization language;
- *STRING_VALUE* – the field is designed to store localized value not longer than 4,000 characters;
- *CLOB_VALUE* – the field is designed to store localized value longer than 4 000 characters.

The dictionary property can be a link to the other dictionary, the relations of the kind are stored in the table *DICT_TOONEREFS*:

- *NAME* – relation name, defines the name of the *property* of generated *class* for *the class type*, which it refers to;
- *CAPTION* – relation name displayed in the screen forms;
- *DICT_OBJ_ID (FK)* – a dictionary, which property is a link to the other dictionary;
- *PROP_OBJ_ID (FK)* – a property, which is a link to the other dictionary. For the type value of this property (*TYPE_ID*), *long* must be selected;

- *REF_DICT_OBJ_ID (FK)* – a dictionary, which specified property refers to.

Besides, through *DICT_TOONEREFS* table, the parent property is set for the tree-type dictionaries. In that case, *DICT_OBJ_ID (FK)* and *REF_DICT_OBJ_ID (FK)* values coincide.

As a result of fill-in of all mandatory attributes and properties in the *subject scheme* of the database, a new table *TABLE_NAME* will be created with primary key *ID* and fields *COLUMN_NAME*, which will be used for storage of data of new dictionary.

Besides, a view is created for each dictionary table, through which all read/write operations are carried out. The view is necessary for:
- versioning of DBMS scheme;
- support of time zones;
- transparent localization of multilanguage properties of the dictionaries.

Let us consider creation of the goods dictionary by the example.

We describe the attributes and properties of new dictionary "Goods". As a result, the following metadata are stored in *the kernel scheme*:

VDICTIONARIES

ID	NAME	CAPTION	TABLE_NAME	CASH
1	Articles	Articles	ARTICLES	FALSE

VDICT_PROPERTIES

ID	NAME	CAPTION	DICT_OBJ_ID	COLUMN_NAME	T
1	Name	Name	1	NAME	
2	ShortName	Short name	1	SHORT_NAME	6
3	Description	Description	1	DESCRIPTION	8
4	Code	Code	1	CODE	3
5	Price	Price	1	PRICE	1

On their basis, GOODS table is created by the kernel in *the subject scheme* of the database.

Table *LINK_TABLES* keeps the attributes, which describe the link table in the *subject scheme* of the database:
- *NAME* – a name of the link table, defines the name of generated *class*;
- *CAPTION* – link table name displayed in the screen forms; the value of this attribute is inserted by default in the corresponding field in case of fill-in of dictionary link to it;
- *TABLE_NAME* – table name in the *subject scheme* of the database.

Table *LINK_PROPERTIES* describes the properties of the link table – in fact, its table fields in the *subject scheme* of the database.
- *NAME* – name of the property, defines the name of the *property* of generated *class*;
- *CAPTION* – a name of the link table property displayed in the screen forms;
- *LINK_OBJ_ID (FK)* – a link to the link table, which the property belongs to, filled in automatically upon its addition to the link table;
- *COLUMN_NAME* – a name of the table field in the *subject scheme* of the database;
- *TYPE_ID (FK)* – property type (see details in section Data types);

- *CONTROL_ID (FK)* – control element, which will be used by default in the screen forms for input of the values of this property, to be selected from those offered by the system. For example, for the string property, it can be string, field or string for input of password hiding the characters;
- *STRING_SIZE* limits the string length of for *TYPE_ID* property type, *string* or *text* is selected;
- *IS_MULTILANGUAGE* – a flag indicating if translation is available for that property;
- *IS_REQUIRED* – a flag indicating if the property is mandatory for fill-in;
- *DEFAULT_VALUE* – a default value, inserted automatically in case of creation of new element of the link table;
- *IS_KEY* – a flag indicating that this property is the key one. Each set of key properties of the link table (marked with *IS_KEY* flag), only one set of values of its other properties corresponds to. Each link table must have at least two key properties.

The link table property can be a link to the dictionary, the relations of the kind are stored in the table *LINK_TOONEREFS*:

- *NAME* – relation name, defines the name of the *property* of generated *class* for *the class type*, which it refers to;
- *CAPTION* – relation name displayed in the screen forms;
- *LINK_OBJ_ID (FK)* – a link table, which property is a link to the dictionary;
- *PROP_OBJ_ID (FK)* – a property of the link table, which is a link to the dictionary. For the type value of this property (*TYPE_ID*), *long* must be selected;
- *REF_DICT_OBJ_ID (FK)* – a dictionary, which specified property refers to.

As a result of fill-in of all mandatory attributes and properties in the *subject scheme* of the database, a new table *TABLE_NAME* will be created with the fields *COLUMN_NAME*, which will be used for storage of data of the link table.

Table *DICT_LINKTABLES* keeps the relations of the dictionary with others using the link tables:

- *NAME* – name of the link table; defines the name for *collection property* of generated *class*;
- *CAPTION* – a name of the link table displayed in the screen forms, inserted by default from the corresponding attribute *LINK_TABLES*, can be modified;
- *DICT_OBJ_ID (FK)* – a link to editable dictionary, filled in automatically;
- *LINK_OBJ_ID (FK)* – a link table, which the editable dictionary will be bound to;
- *PROP_OBJ_ID (FK)* – a link table field, which ID (*ID*) of the properties of editable dictionary will correspond to.

The link tables have no own interface for data editing, as for instance the elements of the dictionaries have. In the screen form for editing of the dictionary records, which we bind using the link tables with others, a tab is created automatically with the name, set in the attribute *CAPTION*, for which the relations will be possible to specify for the elements of current dictionary with the elements, set in the link table *LINK_OBJ_ID (FK),* and to edit their related variables.

Table *DICT_TOMANYREFS* describes relations of the dictionary with others through "many-to-many" relations and their corresponding table in the *subject scheme* of the database:

- *NAME* – a name of the link to many, defines the name for the *collection property* of generated *class*;
- *CAPTION* – a name of the link to many displayed in the screen forms;
- *DICT_OBJ_ID (FK)* – a link to the dictionary being created (edited), filled in automatically;
- *TABLE_NAME* – a name for the tables of links to many in the *subject scheme* of the database;
- *COLUMN_NAME* – a property (field) of the table of links to many, according to which the dictionary being created (edited) will be bound to the other;
- *REF_DICT_OBJ_ID (FK)* – a link to another dictionary, which that being created (edited) is bound to;
- *REF_COLUMN_NAME* – a property (field) of the table of links to many, which the other dictionary will be bound to.

As a result of fill-in of all properties in the *subject scheme* of the database, a new table (of the links to many) will be created with the name *TABLE_NAME* and two fields *COLUMN_NAME* and *REF_COLUMN_NAME*, referring to the key fields of two dictionaries *DICT_OBJ_ID (FK)* and *REF_DICT_OBJ_ID (FK)*.

The tables of "many-to-many" relations have no own interface for data editing in the same manner as the link tables. While binding the current dictionary to the links to many with other, we create thus a tab in its screen form, set in the attribute *CAPTION*, on which the relations of the records of current dictionary can be created with the dictionary records *REF_DICT_OBJ_ID (FK)*.

Localization

System Ultimate AEGIS® supports multilingualism of dictionaries. It means that property of the dictionary can have values in one and more languages. For example, item names in the company price list working with suppliers from Finland can be in the Russian, Finnish and Swedish languages. In this case the most used *language from languages is assigned by default*. Its identifier (*ID* of *LANGUAGES table*) is 1.

The dictionary which has multilingual property is considered localizable (it is set by *IS_MULTILANGUAGE DICT_PROPERTIES table*). Localized values of the dictionary properties (including in language by default) are stored in *PROP_TRANSLATIONS* table in *the kernel scheme of* the database.

All access to the dictionaries that are stored in *the application database scheme* as, for example, reading/record operations, are happened through representations. Including it is made also for implementation of multilingualism.

Localization of the dictionary can be transparent or opaque (it is set by *TRANSP_LOCALIZATION* property of the *DICTIONARIES table*).

Transparent localization

In case of the transparent localization values of the dictionary multilingual properties entered by the user are registered in the table *PROP_TRANSLATIONS* in *the kernel scheme of* the database. Besides, the last actual property value, regardless of that what language it is entered, is replicated also in the table of the dictionary *in the application database scheme*. It allows simplifing search in the change history considerably.

When the user addresses to the transparent localizable dictionary, he gets access to its representation where multilingual properties are provided in the user's language (the user language is defined by the table field *LANG_ID USERS*).

When the transparent localization is in the representation covering the dictionary table, values of the user language will be always on the place of multilingual property. When changes are entered into the dictionary (representation) by the user, values of the user language will be written in the table *PROP_TRANSLATIONS*. Each field requires execution of one operator JOIN. Therefore it is necessary to remember that the transparent localization switching on for the large dictionary can lead to decline in production.

Opaque localization

In case of the opaque localization values of the dictionary multilingual properties entered by the user are registered in the table *PROP_TRANSLATIONS* in *the kernel scheme of the database*. Besides, if property value was entered in language by default (and only in this case), it is also replicated in the table of the dictionary *in the application database scheme*.

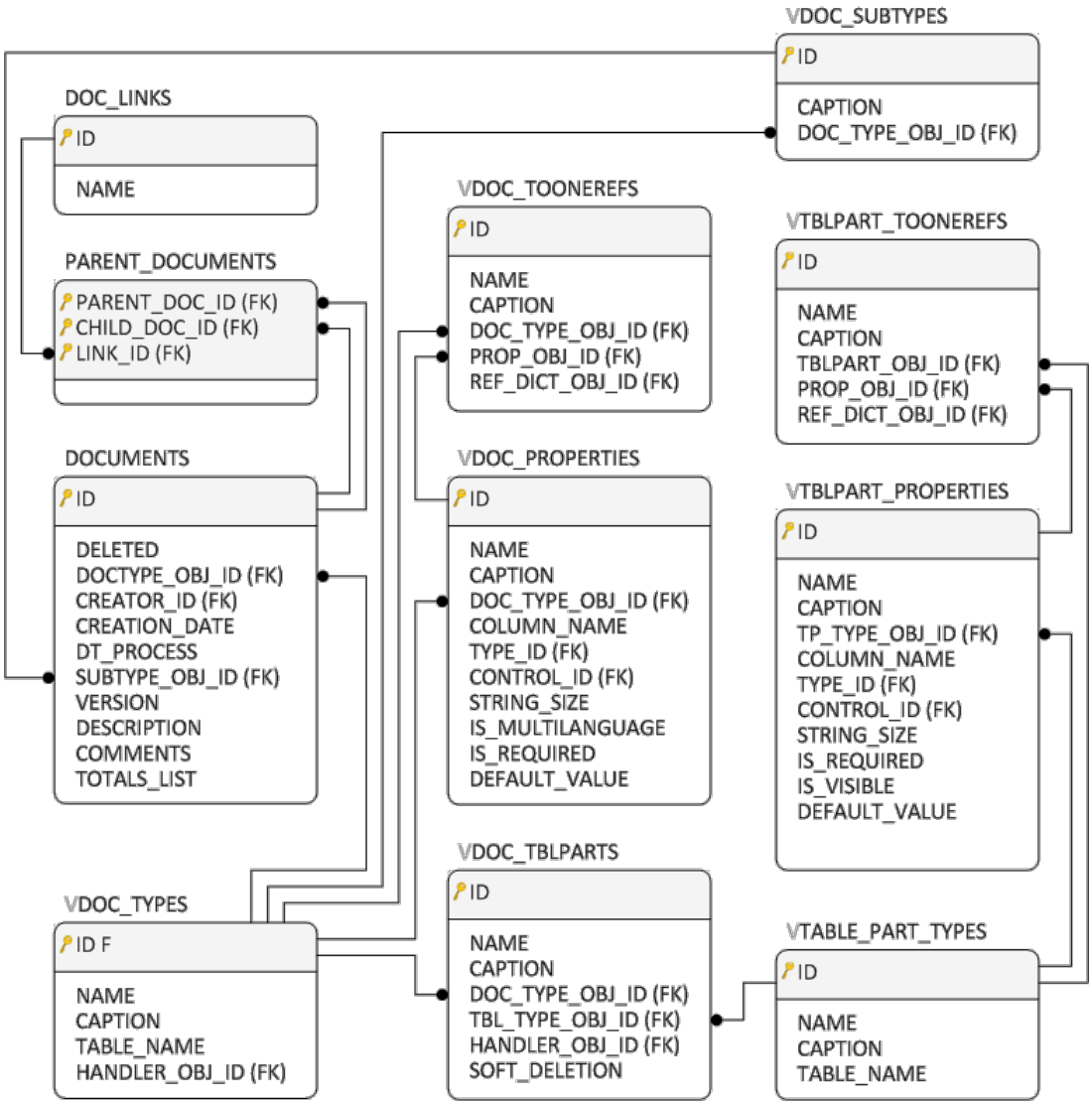
When the user addresses to opaque localizable dictionary, he gets access to its representation where multilingual properties are provided in the language by default.

Documents

The documents consist of the head and one and more table parts, their description is kept in the *kernel scheme* of the database in the following tables:

- *DOC_TYPES* – types of documents;
- *DOCUMENTS* – mandatory (predefined) properties of the heads of documents;
- *DOC_PROPERTIES* – properties of the heads of documents;
- *DOC_TOONEREFS* – relations of the documents properties with the dictionaries;
- *DOC_SUBTYPES* – subtypes of documents;
- *TABLE_PART_TYPES* – table parts of the documents;
- *TBLPART_PROPERTIES* – properties of the table parts of documents;
- *TBLPART_TOONEREFS* – relations of the properties of the document table parts with the dictionaries;
- *DOC_TBLPARTS* – relations of the types of documents and table parts;
- *PARENT_DOCUMENTS* – relations of the parent documents and children documents;
- *DOC_LINKS* – types of the relations of the parent documents and children documents.

Data model looks like as follows:



Tables *DOCUMENTS*, *DOC_LINKS* and *PARENT_DOCUMENTS* of the data model have no prefix "V". They are not views because they are not versioned. For example, the properties of the document head stored

in the table *DOCUMENTS* are similar for all types of documents, they are predefined by the developers of Ultimate AEGIS® system and are unavailable for editing to the application developer.

Let us dwell on each of the tables in details.

The document type assigns a set of fields of the document head, set of its table parts and set of handlers.

Table *DOC_TYPES* keeps the attributes, which describe the document type and its table in the *subject scheme* of the database:

- *NAME* – a name of the document type, defines the name of generated *class*;
- *CAPTION* – document type name displayed in the screen forms, e.g. for the document type Invoice it can be "Receipt" or "Receipts note";
- *TABLE_NAME* – table name in the *subject scheme* of the database. For convenience of work with all tables of the document types, prefix "D_" will be added, for instance, for the value *TABLE_NAME* – INVOICE name of the table in the database will be D_INVOICE;
- *HANDLER_OBJ_ID (FK)* – the handler of events for this document type (before creation, before saving, after saving, before deletion, after deletion), is created as may be necessary.

Table *DOCUMENTS* keeps the values predefined by the developers of the set of fields of document type; besides, the names of the fields define the names of the *properties* of generated *class*, as well as the fields of the table of document type in the *subject scheme* of the database:

- *DELETED* – a flag indicating the document is deleted (*false* – deleted, *true* – not);
- *DOCTYPE_OBJ_ID (FK)* – a link to the document type, which the property belongs to, filled in automatically;
- *CREATOR_ID (FK)* – document creator (id of system user), filled in automatically;
- *CREATION_DATE* – document creation date, filled in automatically;
- *DT_PROCESS* – document processing date;
- *SUBTYPE_OBJ_ID (FK)* – document subtype;
- *VERSION* – ancillary box, filled in automatically using incrementing counter upon each saving of the document. It is used at concurrent editing of the document by two users in order not to let another to save the document after making changes to it by the first one;
- *DESCRIPTION* – automatically generated description of the document of type "{document type} No. {document number} dated {document creation date}";
- *COMMENTS* – document description entered manually;
- *TOTALS_LIST* – a list of transactions automatically generated by the transaction processor based on the document.

If the application developer is short of predefined properties, comprising the head of any new document type, own properties can be described in the table *DOC_PROPERTIES*:

- *NAME* – name of the property, defines the name of the *property* of generated *class*;
- *CAPTION* – a name of the document type property displayed in the screen forms;
- *DOC_TYPE_OBJ_ID (FK)* – a link to the document type, which the property belongs to, filled in automatically when it is added;
- *COLUMN_NAME* – a name of the table field in the *subject scheme* of the database;
- *TYPE_ID (FK)* – property type (see details in section Data types);
- *CONTROL_ID (FK)* – control element, which will be used by default in the screen forms for input of the values of this property, to be selected from those offered by the system. For example, it can be string or field for the string property;
- *STRING_SIZE* limits the string length if for *TYPE_ID* property type *string* is selected;
- *IS_MULTILANGUAGE* – a flag indicating if translation is available for that property;
- *IS_REQUIRED* – a flag indicating if the property is mandatory for fill-in;

- *DEFAULT_VALUE* – a default value, which is inserted automatically in case of creation of new document.

The document property can be a link to the dictionary, the relations of the kind are stored in the table *DOC_TOONEREFS*:

- *NAME* – relation name, defines the name of the *property* of generated *class* for *the class type*, which it refers to;
- *CAPTION* – relation name displayed in the screen forms;
- *DOC_OBJ_ID (FK)* – document type, which property is a link to the dictionary;
- *PROP_OBJ_ID (FK)* – a property of the document type, which is a link to the dictionary. For the type value of this property (*TYPE_ID*), *long* must be selected;
- *REF_DICT_OBJ_ID (FK)* – a dictionary, which specified property refers to.

As a result of filling in of all mandatory attributes and properties in the *subject scheme* of the database, a new table *D_TABLE_NAME* will be created with primary key *ID,* all fields from the table *DOCUMENTS* (*IS_ALIVE, BALANCE_ID (FK), DOCTYPE_OBJ_ID (FK)* etc.) plus the fields *COLUMN_NAME*, which will be used for storage of the head data of all documents corresponding to created type.

Let us consider creation of document type "Income" by the example.

As a result of description of the attributes and properties of new document type in the *kernel scheme*, in the *subject scheme* of the database, table D_INVOICE will be created:

VDOC_TYPES

ID	NAME	CAPTION	TABLE_NAME
1	Invoice	Invoice	D_INVOICE

VDOC_PROPERTIES

ID	NAME	CAPTION	DOC_TYPE_OBJ_ID	COLUMN_NAME
1	Company	Company	1	COMPANY
2	BillTo	Bill to	1	BILL_TO
3	ShipTo	Ship to	1	SHIP_TO
4	Total	Total amount	1	TOTAL
5	Shipping	Shipping amount	1	Shipping

D_INVOICE

ID
IS_ALIVE
BALANCE_ID (FK)
DOCTYPE_OBJ_ID (FK)
CREATOR_ID (FK)
DT_CREATED
DT_PROCESS
SUBTYPE_OBJ_ID (FK)
VERSION
DESCRIPTION
COMMENTS
TOTALS_LIST
COMPANY
BILL_TO
SHIP_TO
TOTAL
SHIPPING

The data of the subject area is stored in the *subject scheme* of the database. That is the heads of all documents created in the system will be stored in the tables of the *subject scheme* corresponding to their type, e.g. head of the orders in the table D_ORDERS, head of invoices in the D_INVOICE, head of reserves in D_RESERVE. But at the same time, all common fields of the heads of all documents, irrespective of their type, will be also replicated in the *kernel scheme* into the table *DOCUMENTS*.

Table *DOC_SUBTYPES* describes the subtypes of documents:

- *CAPTION* – a name of the document subtype displayed in the screen forms;
- *DOC_TYPE_OBJ_ID (FK)* – a link to the document type, which the subtype corresponds to.

Table *TABLE_PART_TYPES* keeps descriptions of the table parts of documents and their tables in the *subject scheme* of the database

- *NAME* – a name of the table part type, defines the name of *string class* of the table part;
- *CAPTION* – a name of the table part tab displayed in the screen forms;
- *TABLE_NAME* – table name in the *subject scheme* of the database. For convenience of work with all tables of the table parts, prefix "TP_" will be added, e.g. for the value *TABLE_NAME* – RESERVE table name in the database will be TP_RESERVE.

Table *TBLPART_PROPERTIES* describes the properties of the table parts of documents – in fact, the table fields of the table part in the *subject scheme* of the database:

- *NAME* – the table part property name defines the name of the *property of string class* of the table part;
- *CAPTION* – a name of the table part property displayed in the screen forms;
- *TP_TYPE_OBJ_ID (FK)* – a link to the table part type, which the property belongs to, filled in automatically when it is added;
- *COLUMN_NAME* – a name of the table field in the *subject scheme* of the database;
- *TYPE_ID (FK)* – property type (see details in section Data types);
- *CONTROL_ID (FK)* – control element, which will be used by default in the screen forms for input of the values of this property, to be selected from those offered by the system. For example, it can be string or field for the string property;
- *STRING_SIZE* limits the string length if for *TYPE_ID* property type *string* is selected;
- *IS_REQUIRED* – a flag indicating if the property is mandatory for fill-in;
- *IS_VISIBLE* – a flag indicating the need to display the property in the table part of the document screen form, it is used for hiding of the ancillary boxes.
- *DEFAULT_VALUE* – a default value, which is inserted automatically in case of creation of new element of table part content.

The document table part property can be a link to the dictionary, the relations of the kind are stored in the table *TBLPART_TOONEREFS*:

- *NAME* – relation name, defines the name of the *string class property* of the table part of *class type*, which it refers to;
- *CAPTION* – relation name displayed in the screen forms;
- *TBLPART_OBJ_ID (FK)* – document table part, which property is a link to the dictionary;
- *PROP_OBJ_ID (FK)* – table part property, which is a link to the dictionary. For the type value of this property (*TYPE_ID*), *long* must be selected;
- *REF_DICT_OBJ_ID (FK)* – a dictionary, which specified property refers to.

In addition to the properties of table parts described by the application developer, there are also properties predefined by the system developers in the same manner as for the document head. They will be added into each table of new type of the table part created in the *subject scheme* of the database:

- *ID* – record ID (primary key), filled in automatically using incrementing counter;
- *DOCUMENT_ID (FK)* – a link to the document, which the table part is bound to, filled in automatically;
- *TP_ENTRY_ID (FK)* – code of table part entry into the document, corresponds to the record in *DOC_TBLPARTS*;
- *IS_ALIVE* – a flag indicating if the content of the table part is deleted (*false* – deleted, *true* – not);
- *FLAG* – a flag available for editing to the end user in the form of checkbox element in the document screen form.

As a result of fill-in of all mandatory attributes and properties in the *subject scheme* of the database, a new table *TP_TABLE_NAME* will be created with mandatory fields plus the fields *COLUMN_NAME*, which will be used for storage of data of new type of the table part of documents.

Table *DOC_TBLPARTS* describes and stores relations of the types of documents and table parts:

- *NAME* – relation name, defines the name of the *collection of string class instances* of the table part;
- *CAPTION* – a name of the table part displayed in the screen forms, inserted by default from the corresponding attribute *TABLE_PART_TYPES*, can be modified;
- *DOC_TYPE_OBJ_ID (FK)* – a link to the document type, filled in automatically;
- *TBL_TYPE_OBJ_ID (FK)* – a link to the document table part type, which is bound to this document type;
- *HANDLER_OBJ_ID (FK)* – transaction processor for particular type of the table part of this type of documents;

- *SOFT_DELETION* – a flag indicating the need in soft deletion of the table part content of the type. If set to value *true*, the content of corresponding table part of this type of documents will remain in the database and be available for restoration in case of deletion. Generally, used for table parts with manually entered content; for automatically generated ones, it is recommended to set the flag to value *false*.

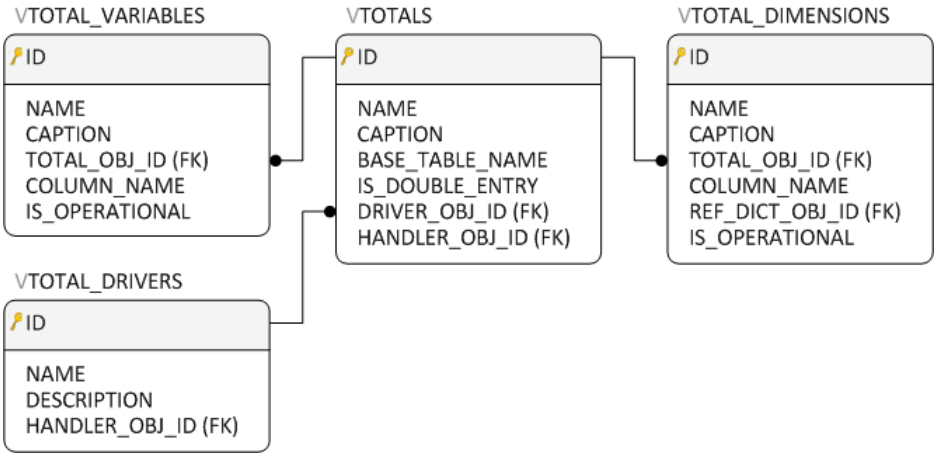
Table *PARENT_DOCUMENTS* keeps the relations of parent documents and children documents:
- *PARENT_DOC_ID (FK)* – a link to the parent document;
- *CHILD_DOC_ID (FK)* – a link to the child document;
- *LINK_ID (FK)* – relation type.

- Table *DOC_LINKS* keeps the types of relations of parent documents and children documents:
- *NAME* – relation type name, serves for description of relation mechanism. For example, "Delivery note is generated on the basis of the order".

Totals

Description of totals is stored *in the kernel scheme* of database in the following tables:
- *TOTALS* – totals;
- *TOTAL_DIMENSIONS* – total dimensions;
- *TOTAL_VARIABLES* – total variables;
- *TOTAL_DRIVERS* – drivers of the totals.

Data model looks like as follows:



Before proceeding with the totals description , let us dwell on their storage structure once more (it was detailed by the example in the first chapter). Every total is implemented using five tables, two of which are *operational* and two *analytical* ones. In the operational table, the information comes immediately upon transaction record in the database, in analytical tables, it comes by the result of totals calculation. The tables can be differentiated by prefixes:
- **TB**_tablename – *operational* summary table (*balance table*), in which only one set of summary values of variables is kept for each set of dimensions;
- **TR**_tablename – *operational* detailed table (*transactions table*), in which all transactions are kept. Having summed their variables for a certain set of dimensions, a summary value can be produced, which is kept in TB_tablename;
- **TD**_tablename – *analytical* detailed table (*detailed transactions table*), in which the transactions are kept, the same as in TR_tablename, but already calculated. To be filled in by the results of totals calculation;
- **TT**_tablename – *analytical* summary table (*detailed transactions balance table*), in which only one set of summary values of variables from the table TD_tablename is kept for each set of dimensions.

Table *TOTALS* keeps the attributes, which describe the total and its tables in the *subject scheme* of the database:

- *NAME* – a name of the total defines the name of generated *class*;
- *CAPTION* – total name displayed in the screen forms;
- *BASE_TABLE_NAME* – a name of operational detailed (RM) table of the total in the *subject scheme* of the database.
  During creation of the total, the application developer creates only one common name of its tables in the *subject scheme* of the database – *TABLE_NAME*. On its basis, the names of all five tables of the total will be defined by addition of corresponding prefix;
- *IS_DOUBLE_ENTRY* – a flag indicating if the total is balance one. if set to value *true*, the double-entry rule will apply in case of changes made to this total;
- *DRIVER_OBJ_ID (FK)* – driver of the total;
- *HANDLER_OBJ_ID (FK)* – handler of events for this total, it is created as may be necessary.

Table *TOTAL_DIMENSIONS* keeps descriptions of the total dimensions – in fact , the fields of its tables:

- *NAME* – a name of the dimension, defines the name for the *property* of generated *class*;
- *CAPTION* – dimension name displayed in the screen forms;
- *TOTAL_OBJ_ID (FK)* – a link to the total, which the dimension belongs to, filled in automatically;
- *COLUMN_NAME* – a name of the table field in the *subject scheme* of the database;
- *REF_DICT_OBJ_ID (FK)* – a link to the Dictionary, which elements represent the total dimension;
- *IS_OPERATIONAL* – a flag indicating if the dimension is operational. if set to value *true*, the dimension value will be always filled in when the data are recorded into the total table. For analytical dimension (a flag is set to value *false*), the value can be ignored and filled in only in case of totals calculation.

Table *TOTAL_VARIABLES* keeps descriptions of the total variables – if fact, the fields of its tables:

- *NAME* – a name of the variable, defines the name for the *property* of generated *class*;
- *CAPTION* – variable name displayed in the screen forms;
- *TOTAL_OBJ_ID (FK)* – a link to the total, which the variable belongs to, filled in automatically;
- *COLUMN_NAME* – a name of the table field in the *subject scheme* of the database;
- *IS_OPERATIONAL* – a flag indicating if the variable is operational. if set to value *true*, the variable value will be always filled in when the data is recorded into the total table. For analytical variable (a flag is set to value *false*), the value can be ignored and filled in only in case of totals calculation.

All created variables of the totals have *decimal* type.

In addition to the dimensions and variables, described by the application developer, every totals has also the system properties predefined by the developers:

- *DELTA_NO* – transaction number;
- *DELTA_DUB_NO* – additional number of the transaction, used during record of the calculated transactions, which can be split into several components;
- *PAIR_TOTAL_ID (FK)* – a link to the other total, which the changes were made to with the same transaction in case of application of double-entry rule;
- *HANDLER_TOTAL_OBJ_ID (FK)* – transaction processor, generated by this transaction;
- *VERSION_ID (FK)* – version of the transaction processor;
- *LOT_NO* – lot number, formed on the basis of the document processing date, document number and transaction number;
- *DOCUMENT_ID (FK)* – a link to the document, that originated the transaction;
- *DT_PROCESS* – processing date for this document.

The below table shows entry of described predefined properties into the tables of the total in the *subject scheme* of the database:

table field	entry of the field into the total table			
	TR	TB	TD	TT
DELTA_NO	+		+	
DELTA_DUB_NO			+	
LOT_NO			+	+
DOCUMENT_ID (FK)	+		+	
DT_PROCESS	+		+	
PAIR_TOTAL_ID (FK)	+		+	
HANDLER_TOTAL_OBJ_ID (FK)	+			
VERSION_ID (FK)	+			

As a result of filling in of all mandatory attributes and properties in the *subject scheme* of the database, five new tables will be created *TR_TABLE_NAME*, *TB_TABLE_NAME*, *TD_TABLE_NAME* and *TT_TABLE_NAME* with mandatory fields, according to the above table, plus the fields *COLUMN_NAME* of dimensions and variables, which will be used for storage of data of new total.

Let us consider creation of the total "Remaining stock at the warehouse".

As a result of description of the attributes and properties of new total in the *kernel scheme*, in the *subject scheme* of the database 5 corresponding tables will be created:

VTOTALS

ID	NAME	CAPTION	BASE_TABLE_NAME	IS
1	BalancesWh	Warehouse balances	BALANCES_WH	1

VTOTAL_DIMENSIONS

ID	NAME	CAPTION	TOTAL_OBJ_ID	COLUMN_NAME	R
1	Good	Good	1	GOOD	1
2	Warehouse	Warehouse	1	WAREHOUSE	6

VTOTAL_VARIABLES

ID	NAME	CAPTION	TOTAL_OBJ_ID	COLUMN_NAME	R
1	Quantity	Quantity	1	QUANTITY	1
2	Amount	Amount	1	AMOUNT	6

TB_BALANCES_WH

GOOD
WAREHOUSE
QUANTITY
AMOUNT

TD_BALANCES_WH

DELTA_NO
DELTA_SUB_NO
LOT_NO
DOCUMENT_ID (FK)
DT_PROCESS
PAIR_TOTAL_ID (FK)
GOOD
WAREHOUSE
QUANTITY
AMOUNT

TR_BALANCES_WH

DELTA_NO
DOCUMENT_ID (FK)
DT_PROCESS
PAIR_TOTAL_ID (FK)
HANDLER_TOTAL_OBJ_ID (FK)
VERSION_ID (FK)
GOOD
WAREHOUSE
QUANTITY
AMOUNT

TT_BALANCES_WH

LOT_NO
GOOD
WAREHOUSE
QUANTITY
AMOUNT

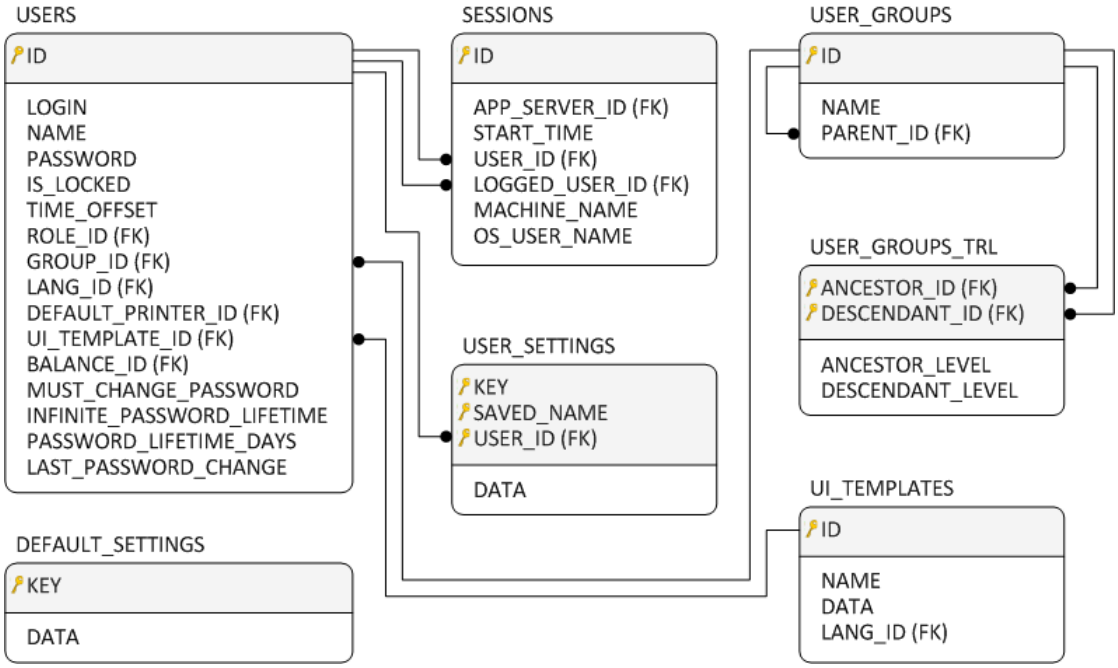
Table *TOTAL_DRIVERS* keeps descriptions of the drivers of the totals:

- *NAME* – a name for the driver of the total;
- *DESCRIPTION* – driver name displayed in the screen forms;
- *HANDLER_OBJ_ID (FK)* – driver handler.

Users

The kernel of Ultimate AEGIS® system already has required procedures for authorization and verifying user rights.

Information on users is stores in the *scheme* of the database kernel in the form of following tables:



USERS table contains the data on users:

- *LOGIN* – login to enter the system;
- *NAME* – name of user used by the system for calling the user;
- *PASSWORD* – password hash sum, under which the sign-in is carried out in connection with the login;
- *IS_LOCKED* – flag indicating that the user account is locked; if *true*, log-in is impossible;
- *TIME_OFFSET* – user's time zone;
- *ROLE_ID (FK)* – user's role;
- *GROUP_ID (FK)* – group that the user is included to;
- *LANG_ID (FK)* – interface language;
- *DEFAULT_PRINTER_ID (FK)* – default printer;
- *UI_TEMPLATE_ID (FK)* – user interface template;
- *BALANCE_ID (FK)* – balance code used to store data on several companies in a single database. The balance code specified will be provided automatically, if user is allowed to select it;
- *MUST_CHANGE_PASSWORD* – flag indicating that it is needed to change password the next time the user logs in;
- *INFINITE_PASSWORD_LIFETIME* – flag indicating that the password lifetime is infinite (*true* – password lifetime infinite, *false* – finite);
- *PASSWORD_LIFETIME_DAYS* – number of days that the password lifetime is limited to; used together with the *INFINITE_PASSWORD_LIFETIME* flag, if set in *false*;
- *LAST_PASSWORD_CHANGE* – date of last password change. This is used (together with *PASSWORD_LIFETIME_DAYS*) in calculation of a date, when the user will be offered to change his password, while logging in.

Users are arranged in groups in a tree-like structure; description of the structure is in the *USER_GROUPS* table:

- *NAME* – name of group;
- *PARENT_ID (FK)* – link to parent group.

For ease of reference, the *USER_GROUPS* table's contents is automatically copied to the *USER_GROUPS_TRL* table, which contains a list of all children for each parent:

- *ANCESTOR_ID (FK)* – link to parent group;
- *DESCENDANT_ID (FK)* – link to child group;
- *ANCESTOR_LEVEL* – parent group level;
- *DESCENDANT_LEVEL* – child group level.

How to fill in the *USER_GROUPS_TRL* table is shown below:

USER_GROUPS

ID	NAME	PARENT_ID
1	Moscow	NULL
2	Main office	1
3	Sales department	2
4	Warehouse	2
5	Warranty department	2
6	Accounts department	2
7	Retail store	1

USER_GROUPS_TRL

ANCESTOR_ID	DESCENDANT_ID	ANCESTOR_LEVEL	DESCENDANT_LEVEL
1	2	1	2
1	7	1	2
1	3	1	3
1	4	1	3
1	5	1	3
1	6	1	3
2	3	2	3
2	4	2	3
2	5	2	3
2	6	2	3

UI_TEMPLATES stores configuration of user interfaces (menu structure and sets of client application screen forms), which is as flexible-adjusted as one may wish:

- *NAME* – name of user interface;
- *DATA* – interface configuration;
- *LANG_ID (FK)* – interface language.

USER_SETTINGS table stores the settings of client application screen forms (unlike the *UI_TEMPLATES* table, this stores not a list of screen forms, but their configuration, e. g., arrangement and size of windows, a list and succession of columns, etc.):

- *KEY* – key; as a rule, this is a screen form name;
- *SAVED_NAME* – name of settings; usually, used to describe their logic;
- *USER_ID (FK)* – user, who made the settings;
- *DATA* – screen form settings.

DEFAULT_SETTINGS stores the default settings of screen forms; if no changes were made to the settings by user, they are loaded from this table, not from *USER_SETTINGS*:

- *KEY* – key; as a rule, this is a screen form name;
- *DATA* – screen form settings.

When authorizing a user, a session is created; the session parameters are stored in the *SESSIONS* table:

- *APP_SERVER_ID (FK)* – application server that the client application worked with;
- *START_TIME* – session start time;
- *USER_ID (FK)* – user authorized (his login and password were used);

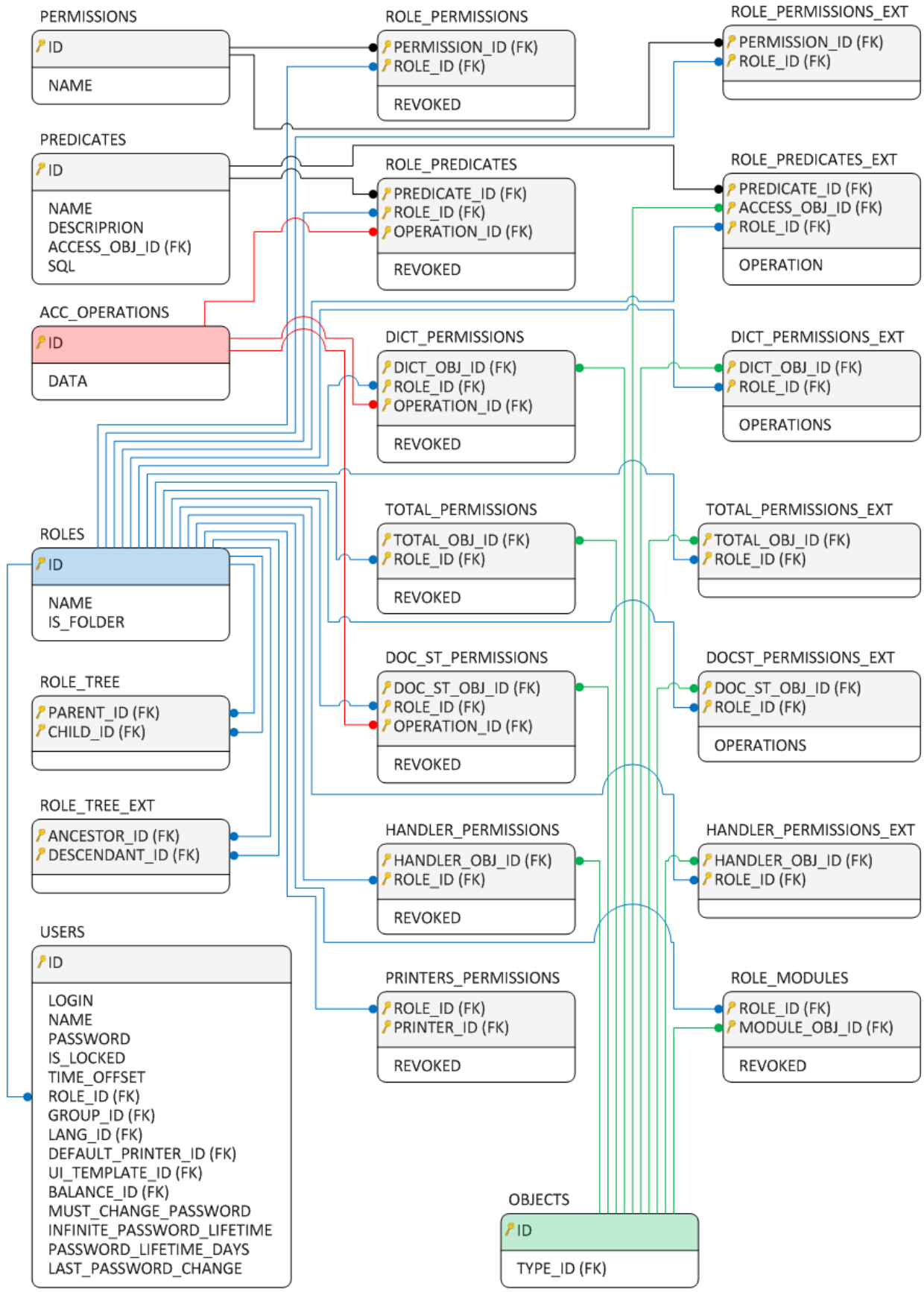
- *LOGGED_USER_ID (FK)* – user, which login was entered when signing in with the option "Sign in with a different account" (details on this option is in the "Starting application" section of the user's manual).
- *MACHINE_NAME* – name of computer in the operating system, on which the client application was launched;
- *OS_USER_NAME* – user name, under which he signed in to the operating system, under which the client application was launched.

User permissions

The user permissions are created using the *roles*, *permissions* and *predicates:*

- By means of *roles* access to system objects is provided: to dictionaries, results, handlers, documents of specific subtype, modules of screen logic and printers. The roles are arranged as follows:
  - permissions and predicates are bound to roles;
  - the roles are structured in the tree-type dictionary;
  - the tree of roles has structure many to many: the role can have many descendants and many parents;
  - the parent role has all permissions provided by the child role including permissions and predicates.
- *The predicates* are used if access provided with the role to the elements of the objects like dictionary, document or total should be limited. If these limitations can be expressed with SQL-query – a predicate will apply to them. For example, the sales manager should be provided with a possibility to work only with retail clients but not with wholesale and corporate ones.
- *By permissions* situations are regulated, when it is necessary to limit access to any actions: there is a permission – it is possible to perform an action, there is no permission – it is impossible. In fact, these are control points in the program code, in which the user is checked for having corresponding permissions.

A model of data implementing the mechanisms of the user permissions looks like as follows:



System table *ACC_OPERATIONS* keeps a list of operations available for system objects:

- *ID* – operation ID, it is also its number in the bit mask;

- *NAME* – operation name.

A list of operations kept in the table *ACC_OPERATIONS*:

id	1	2	4	8
operation	read	add	modify	delete

Operation ID is the number of its bit in the bit mask at the same time. If the operation on the object is set with a bit mask, e.g. all specified operations, except for deletion, are permitted for its value "0111", and the value "0001" grants read access only.

Roles

Table *ROLES* keeps the list of roles:

- *NAME* – role name;
- *IS_FOLDER* – a flag indicating if the role is folder. The folders are intended for convenience of the roles view and are used solely for their grouping. The functionality of the folder roles is restricted to a number of prohibitions:
  - a permit for access to the object cannot be issued using the folder-role;
  - a folder-role cannot be assigned to the user;
  - a folder-role cannot be made a child for other role if not the folder either.

The roles are arranged into the tree structure, which description is kept in the table *ROLES_TREE*:

- *PARENT_ID (FK)* – parent role;
- *CHILD_ID (FK)* – child role.

For convenience of work, the content of table *ROLES_TREE* is replicated automatically into the table *ROLES_TREE_EXT*, where a list of all children is kept for each parent:

- *ANCESTOR_ID (FK)* – a link to the parent role;
- *DESCENDANT_ID (FK)* – a link to the child role.

The roles represent essentially a combination of users, united with similar rules. That is, the same role is designed for the users with similar functionality. The parent role provides the user also with all the permissions set with its children roles. For example, the roles defining the functionality of subordinates, being the children to the department head role, provides them with a possibility to perform all operations available for them:

Therefore, a task for defining of user permissions with high-level parent role will reduce also to defining the permissions of all of its children roles. The more levels are present in the roles tree, the more labour-intensive task it is. For optimization of these queries, the very table *ROLES_TREE_EXT*, is designed, where all child roles are specified explicitly for each role, irrespective of their nesting level. The mechanisms for user permissions checks, implemented with the kernel, work just with such optimized tables, which are filled in automatically, they can be differentiated by suffix *_EXT* in the name.

Permissions

Table *PERMISSONS* keeps a list of permissions:

- *ID* – permission ID, according to which the checking is carried out at the control point of the program code;
- *NAME* – permission name.

Table *ROLE_PERMISSONS* keeps a list of permissions bound to roles:

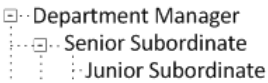
- *PERMISSON_ID (FK)* – a permission, being bound to the role;
- *ROLE_ID (FK)* – a role, which the permit is bound to;
- *REVOKED* – revocation of permission binding to the role. It is used for revocation of permission binding derived by the parent role from the child.

The parent role has all permissions provided with its children roles. The same refers to permissions. But the situations occur when the child role has a permission but the parent role does not need it. For example, the storekeeper role has bound permission for performance of dispatch of goods from the storehouse. At the same time, it is a child of the role of the storehouse coordinator. But the storehouse coordinator is not an accountable officer and does not have the right to do such shipment. The storehouse coordinator parent role can be divested of such permission using two methods:

- to take the permission away from the storekeeper child role but its users will be then divested of corresponding rights;
- to bind the permission also to the parent role but set *REVOKED* property for this relation to value *true*, the child role will then have this permission and the parent role will not.

In the same manner as parent role has permission of its children roles, it has revocations of their permissions too.

For example, while binding a permission to the role of Junior Subordinate, we issue it thus to the parent roles too – Senior Subordinate and Department Head:



Having revoked this permission for the role of Senior Subordinate, we revoke it thus for the Department Head too.

For convenience of work, the content of table *ROLE_PERMISSIONS* is replicated automatically into the table *ROLE_PERMISSONS_EXT*, where a list of all role permissions specified explicitly is kept for each role, taking into account that the parent role has permissions of the children roles if they are not revoked for it with the *REVOKED* property:

- *PERMISSION_ID (FK)* – a link to the permission;
- *ROLE_ID (FK)* – a link to the role.

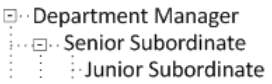
Having bound a permission to the role of Junior Subordinate, we add thus one record into the table *ROLE_PERMISSIONS* and three records into the table *ROLE_PERMISSIONS_EXT*:

ROLES

ID	NAME	IS_FOLDER
1	Department Manager	NULL
2	Senior Subordinate	NULL
3	Junior Subordinate	NULL

PERMISSIONS

ID	NAME
1	Permission



ROLE_PERMISSIONS

PERMISSION_ID	ROLE_ID	REVOKED
1	3	0

ROLE_PERMISSIONS_EXT

PERMISSION_ID	ROLE_ID
1	3
1	2
1	1

Having revoked this permission for the role of Senior Subordinate, we revoke it thus for the Department Head too, while adding one more record into the table *ROLE_PERMISSIONS* and having deleted two records in the table *ROLE_PERMISSIONS_EXT*:

ROLE_PERMISSIONS

PERMISSION_ID	ROLE_ID	REVOKED
1	3	0
1	2	1

ROLE_PERMISSIONS_EXT

PERMISSION_ID	ROLE_ID
1	3

Predicates

Table *PREDICATES* keeps a list of predicates:

- *NAME* – predicate name;
- *DESCRIPTION* – predicate description;
- *ACCESS_OBJ_ID (FK)* – a link to the object, the access to which is limited by the predicate;
- *SQL* – SQL-query, which limits access.

Table *ROLE_PREDICATES* keeps the list of predicates bound to the roles:

- *PREDICATE_ID (FK)* – predicate, being bound to the role;
- *ROLE_ID (FK)* – a role, which the predicate is bound to;
- *OPERATION_ID (FK)* – an operation, which can be performed on elements of the object, the access to which is limited by the predicate;
- *REVOKED* – revocation of predicate binding to the role. It is used for revocation of predicate binding derived by the parent role from the child.

The role, which the predicate is bound too, must have permissions to the object, the access to which is limited by this predicate. For example, if the user needs to be provided with limited access to the contractors' dictionary, the following is required:

- to provide this user role with access to the contractors' dictionary;
- to create a predicate imposing limitations on access to the contractors' dictionary;
- to bind it to the user's role.

Furthermore, the operation, which the limitations are imposed on, must be permitted for the role on this object. For example, binding of the predicate, which imposes limitations on the operation for

deletion of the contractors' dictionary records, to the role, which is granted only with read access, addition and modification of the records of this dictionary, will be ineffective since an attempt will be made to limit access for operation, which is not permitted anyway.

For convenience of work, the content of table *ROLE_PREDICATES* is replicated automatically into the table *ROLE_PREDICATES_EXT*, where a list of all role predicates specified explicitly is kept for each role, taking into account that the parent role has predicates of the children roles if they are not revoked for it with the *REVOKED* property:

- *PREDICATE_ID (FK)* – a link to the predicate;
- *ACCESS_OBJ_ID (FK)* – a link to the object, the access to which is limited by the predicate;
- *ROLE_ID (FK)* – a link to the role;
- *OPERATIONS* – a bit mask, containing the list of operations, which can be performed on the object elements, the access to which is limited by the predicate.

All of previously described properties of the permissions are executed for the predicates too:

- the parent role obtains the same predicates as its children roles;
- revocation of the predicate for the child role revokes it for the parent too.

Permissions to the dictionaries

Table *DICT_PERMISSIONS* keeps the list of dictionary operations permitted for the roles:

- *DICT_OBJ_ID (FK)* – a dictionary, access to which is given by the role;
- *ROLE_ID (FK)* – a link to the role;
- *OPERATION_ID (FK)* – an operation, which can be performed on a dictionary;
- *REVOKED* – revocation of the operation of the role on a dictionary. It is used for revocation of access derived by the parent role from the child.

For convenience of work, the content of table *DICT_PERMISSIONS* is replicated automatically into the table *DICT_PERMISSIONS_EXT*, where a list of all available operations on the dictionaries specified explicitly is kept for each role, taking into account that the parent role has access permissions of the children roles if they are not revoked for it with the *REVOKED* property:

- *DICT_OBJ_ID (FK)* – a dictionary, access to which is given by the role;
- *ROLE_ID (FK)* – a link to the role;
- *OPERATIONS* – a bit mask, containing the list of operations, which can be performed on a dictionary, the access to which is given by the role.

Permissions to the totals

Table *TOTAL_PERMISSIONS* keeps the list of totals available for the roles:

- *TOTAL_OBJ_ID (FK)* – a total, the access to which is given by the role;
- *ROLE_ID (FK)* – a link to the role;
- *REVOKED* – revocation of access to the total for the role. It is used for revocation of access derived by the parent role from the child.

For convenience of work, the content of table *TOTAL_PERMISSIONS* is replicated automatically into the table *TOTAL_PERMISSIONS_EXT*, where a list of all available totals specified explicitly is kept for each role, taking into account that the parent role has access permissions of the children roles if they are not revoked for it with the *REVOKED* property:

- *TOTAL_OBJ_ID (FK)* – a total, the access to which is given by the role;
- *ROLE_ID (FK)* – a link to the role.

Permissions to the documents

Table *DOC_ST_PERMISSIONS* keeps the list of operations on the documents of specified subtype, which are permitted for the roles:

- *DOC_ST_OBJ_ID (FK)* – a subtype of the documents, the access to which is given by the role;
- *ROLE_ID (FK)* – a link to the role;
- *OPERATION_ID (FK)* – an operation, which can be performed on a document of the subtype;
- *REVOKED* – revocation of the operation on the subtype document for the role. It is used for revocation of access derived by the parent role from the child.

For convenience of work, the content of table *DOC_ST_PERMISSIONS* is replicated automatically into the table *DOC_ST_PERMISSIONS_EXT*, where a list of all available operations on the documents of subtypes specified explicitly is kept for each role, taking into account that the parent role has access permissions of the children roles if they are not revoked for it with the *REVOKED* property:

- *DOC_ST_OBJ_ID (FK)* – a subtype of the documents, the access to which is given by the role;
- *ROLE_ID (FK)* – a link to the role;
- *OPERATIONS* – a bit mask, containing the list of operations, which can be performed on the subtype documents, the access to which is given by the role.

Permissions to launch of handlers

Table *HANDLER_PERMISSIONS* keeps the list of handlers available for the roles:

- *HANDLER_OBJ_ID (FK)* – a handler, which launch is permitted by the role;
- *ROLE_ID (FK)* – a link to the role;
- *REVOKED* – revocation of access to the handler for the role. It is used for revocation of access derived by the parent role from the child.

For convenience of work, the content of table *HANDLER_PERMISSIONS* is replicated automatically into the table *HANDLER_PERMISSIONS_EXT*, where a list of all available handlers specified explicitly is kept for each role, taking into account that the parent role has access permissions of the children roles if they are not revoked for it with the *REVOKED* property:

- *HANDLER_OBJ_ID (FK)* – a handler, which launch is permitted by the role;
- *ROLE_ID (FK)* – a link to the role.

Other permissions

Table *ROLE_MODULES* keeps the list of modules of client applications available for the roles:

- *ROLE_ID (FK)* – a link to the role;
- *MODULE_OBJ_ID (FK)* – module of the client application which start is allowed by the role;
- *REVOKED* – revocation of access to the module of client application for the role. It is used for revocation of access derived by the parent role from the child.

- Table *PRINTERS_PERMISSIONS* keeps the list of handlers available for printers:
- *ROLE_ID (FK)* – a link to the role;
- *DICT_OBJ_ID (FK)* – a printer, access to which is allowed by the role;
- *REVOKED* – revocation of access to the handler for printer. It is used for revocation of access derived by the parent role from the child.

Permissions check

All permissions except for those formulated by the developer, are checked at the database level. The developer's permissions can be checked also at the level of a client or application server.

After user authorization at the point of session initiation, the entire set of its permissions is saved in the database context. It can be checked using *PACK_AUTH* pack.

Modules of client applications

Descriptions of client applications and their modules are kept in the *kernel scheme in* the following tables:

- *CLIENT_APPS* – client applications;
- *CLIENT_MODULES* – modules of applications;
- *CLIENT_APP_MODULES* – relations of modules and client applications;
- *CLIENT_ASSEMBLIES* – assemblies of the modules.

Data model looks like as follows:

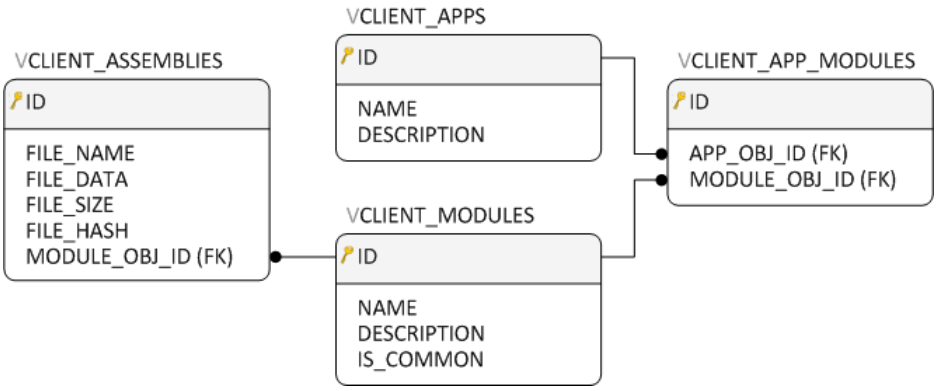


Table *CLIENT_APPS* keeps the list of client applications. For example, an application can be available for desktop PC, application for touch-screen PC or for mobile device:

- *NAME* – application name;
- *DESCRIPTION* – application description.

Table *CLIENT_MODULES* keeps the list of client application modules:

- *NAME* – module name;
- *DESCRIPTION* – module description;
- *IS_COMMON* – flag, indicating the need to give the rights for access to the module. If it is established in *true* value , the user should not have special rights to start the module.

Table *CLIENT_APP_MODULES* keeps the relations of modules and client applications. Several modules can correspond to each client application in the same manner as several applications can correspond to each module:

- *APP_OBJ_ID (FK)* – a module linked to the client application;
- *MODULE_OBJ_ID (FK)* – an application, which the module is linked to.

In terms of implementation, each module represents one and more libraries. A combination of these libraries is called module assembly and is described in the table *CLIENT_ASSEMBLIES*:

- *FILE_NAME* – library file name, which is included into the assembly;
- *FILE_DATA* – library file;
- *FILE_SIZE* – size of the library file in bytes;
- *FILE_HASH* – file hash;
- *MODULE_OBJ_ID (FK)* – a module, which the library is linked to.

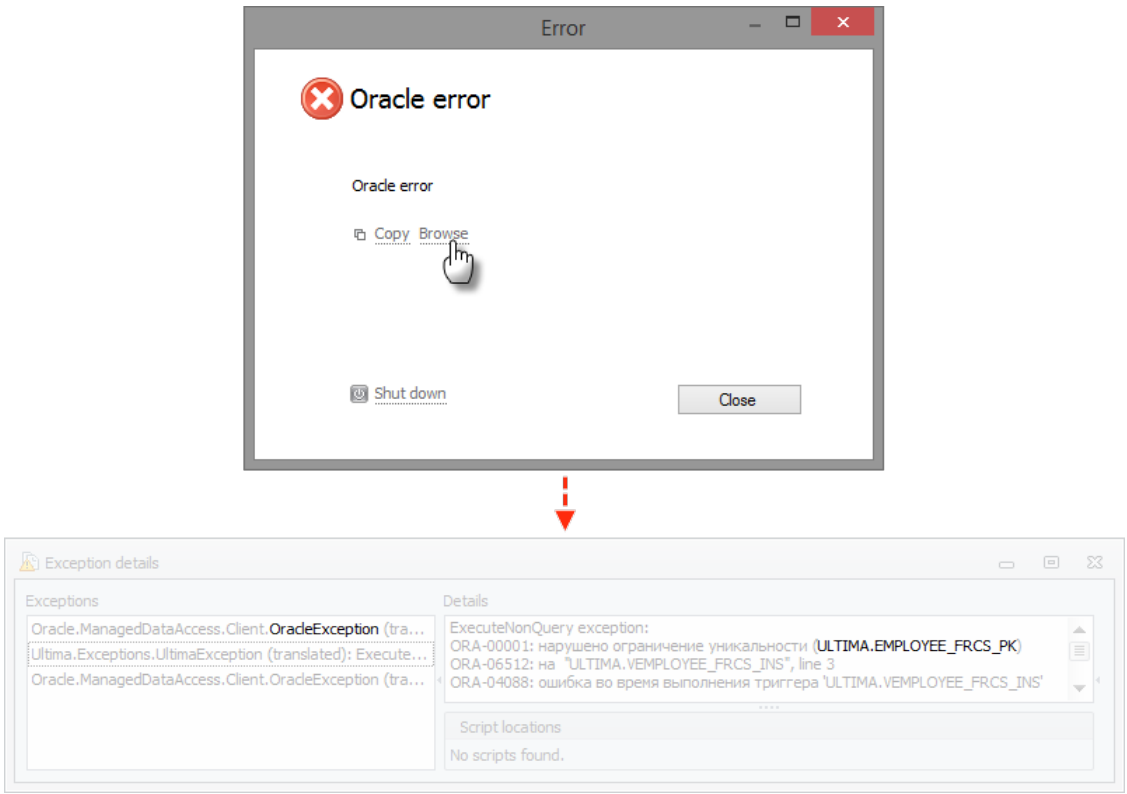
Localization of exceptions

The mechanism for localization of exception implements dispatching reasonable warnings and error messages to the system user in their language.

The data model is represented with the table *EXCEPTION_TRANSLATIONS*, which keeps the list of localized values of exceptions:

- *DESCRIPTION* – localization description;
- *TYPE_PATTERN* – exception type;
- *MESSAGE_PATTERN* – exception text;
- *USE_REGEXP* – a flag indicating the use of regular expressions (detailed description of regular expressions can be found on MSDN website eng/rus) in the fields *TYPE_PATTERN* and *MESSAGE_PATTERN*;
- *TEXT* – localized text for exceptions of corresponding type (*TYPE_PATTERN*) and text (*MESSAGE_PATTERN*);
- *LANG_ID (FK)* – localization language;
- *SORT_INDEX* – sort index of localizations. For two similar pairs of text-type localizations, localization with large index value will be used.

Let us consider functioning of the mechanism of exception localization by the example:

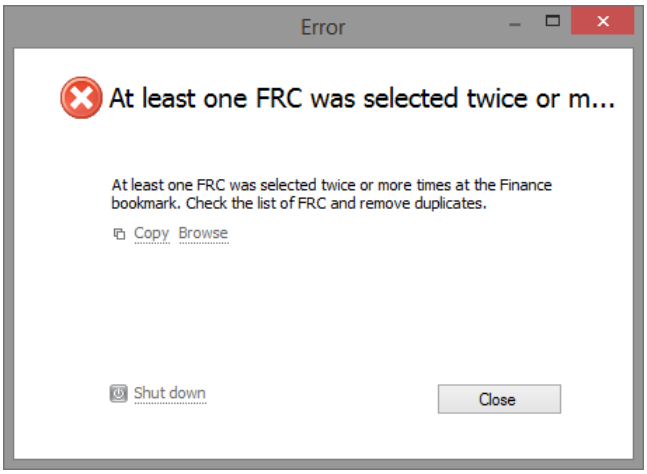


In order to localize this exception, we add a record corresponding to its type (*Exception*) and text (*Message*) into the table *EXCEPTION_TRANSLATIONS*:

DESCRIPTION	ApplicationException | Constant is not found
TYPE_PATTERN	System.ApplicationException
MESSAGE_PATTERN	Constant is not found
USE_REGEXP	0
TEXT	Constant is not found. Copy error text and send it to the developers to dev@company.com
LANG_ID	1

SORT_INDEX	1

As a result, the error message looks like as follows:



If exception has enclosed exceptions, only the first one is localized:

Exception 1 Translated Exception 1
- Exception 2 => - Exception 2
  - Exception 3 - Exception 3

If exception has non-serialized exception among enclosed ones, all exceptions is localized:

Exception 1 Translated Exception 1
- Non Serializable Exception 2 => - Translated Non Serializable Exception 2
  - Exception 3 - Translated Exception 3

Logging

Ultimate AEGIS® system provides the application developer with an option to store the changes made to the tables of the *subject scheme* of the database. The change logs are kept in the *kernel scheme* because one mechanism performs logging in both schemes of the database. The following tables are designed for these purposes:

- *LOG_TABLES* – logged tables;
- *LOG_FIELDS* – logged fields of the tables;
- *LOG_CHANGES* – information about changes made with the rows of logged tables;
- *LOG_FIELDS_CHANGED* – information about the changes made to the logged fields;
- *STAT_COMMAND_EVENTS* – information about calls of the methods of handlers.

Data model looks like as follows:

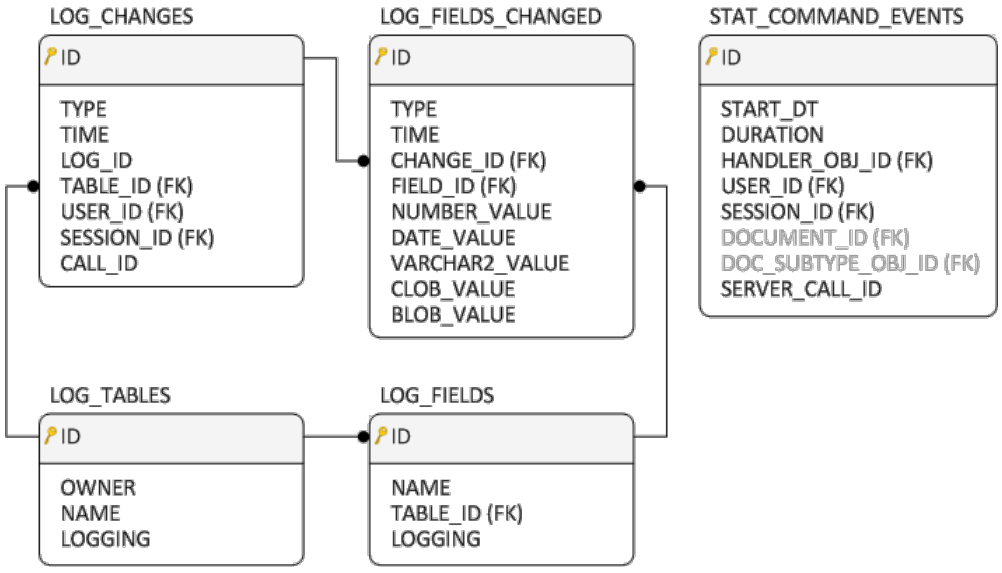


Table *LOG_TABLES* keeps the list of logged tables:

- *OWNER* – database scheme. It can be *kernel scheme* (*kernel*) or *subject scheme* (*ultima*). Only the tables of the *subject scheme* are available to the application developer for logging, therefore value *ultima* is automatically set to this property;
- *NAME* – a name of logged table;
- *LOGGING* – a flag indicating if the table is logged. The flag can be set to value *false*, if the table was logged previously but this option was disabled for it afterwards.

In case of enabling of the logging option for the table, additional ancillary box *LOG_ID* is created automatically.

Table *LOG_FIELDS* keeps the list of logged table fields. The logging mechanism allows storing all changes not for all columns but only for selected ones:

- *NAME* – logged field name;
- *TABLE_ID (FK)* – logged table, which the field corresponds to, filled in automatically;
- *LOGGING* – a flag indicating if the field is logged. The flag can be set to value *false*, if the field was logged previously but this option was disabled for it afterwards.

Table *LOG_CHANGES* keeps the information about the nature of changes, made to the row of logged table, as well as by whom and when they were made (filled in automatically):

- *TYPE* – the type of change made to the table row, can have the following values:
  - *I* – add a row;
  - *U* – change a row;
  - *D* – delete a row;
- *TIME* – time of the changes made to the table row;
- *LOG_ID* – ID of changed row of logged table. It is assigned also to the changed table row itself in the *subject scheme* of the database (for that purpose similar field *LOG_ID* was created in it). ID of changed row is unique and, being assigned to it for the first time, does not change with subsequent changes made to the content of its fields;
- *TABLE_ID (FK)* – logged ID, which row the changes were made to;
- *USER_ID (FK)* – a user that made the changes;
- *SESSION_ID (FK)* – a session (client-server), during which the changes were made;
- *CALL_ID* – ID of the call (method), which preserves constant value after login to the application server and till logout.

Table *LOG_FIELDS_CHANGED* keeps the changes made to the field of logged table (filled in automatically):

- *TYPE* – type of logged data, defines in which field of the table *LOG_FIELDS_CHANGED* , the changes will be kept, being made to the content of logged fields of the table. It can have the following values:
  - *N* corresponds to the field *NUMBER_VALUE*;
  - *D* corresponds to the field *DATE_VALUE*;
  - *V* corresponds to the field *VARCHAR2_VALUE*;
  - *C* corresponds to the field *CLOB_VALUE*;
  - *B* corresponds to the field *BLOB_VALUE*;
- *TIME* – time of the changes made to the field of logged table;
- *CHANGE_ID (FK)* – change made to the row of logged table;
- *FIELD_ID (FK)* – logged field of the table, the changes were made to;
- *NUMBER_VALUE* – a field intended for storage of changes made to the content of numeric type logged field (*long*, *decimal* or *bool*);
- *DATE_VALUE* – a field intended for storage of changes made to the content of logged field of the type (*date* or *DateTime*);
- *VARCHAR2_VALUE* – a field intended for storage of changes made to the content of logged field of string type (*string*, *text* and *LargeText* not longer than 4,000 characters);
- *CLOB_VALUE* – a field intended for storage of changes made to the content of logged field of string type (*LargeText* longer than 4,000 characters);
- *BLOB_VALUE* – a field intended for storage of changes made to the content of logged field of the type *byte[]*.

Let us consider functioning of the logging mechanism by the example for the dictionary "Contractors":

AGENTS

ID	NAME	EMAIL	PHONE
1	Ken Kesey	kenk@gmail.com	89262345124
2	John Steinbeck	jhons@gmail.com	89031894536
3	Kurt Vonnegut	kurtv@yahoo.com	89105238734
4	Harper Lee	harperl@gmail.com	89015463738
5	Jerome Salinger	jeromes@yahoo.com	89269548736

If logging is enabled for its fields NAME, EMAIL and PHONE, in the *kernel scheme* in the tables LOG_TABLES and LOG_FIELDS corresponding records will be added, and for the table AGENTS of *the subject scheme* additional field LOG_ID will be created:

LOG_TABLES

ID	OWNER	NAME	LOGGING
5	ultima	AGENTS	1

LOG_FIELDS

ID	NAME	TABLE_ID	LOGGING
17	NAME	1	1
18	EMAIL	1	1
19	PHONE	1	1

AGENTS

ID	NAME	EMAIL	PHONE	LOG_ID
1	Ken Kesey	kenk@gmail.com	89262345124	
2	John Steinbeck	jhons@gmail.com	89031894536	
3	Kurt Vonnegut	kurtv@yahoo.com	89105238734	
4	Harper Lee	harperl@gmail.com	89015463738	
5	Jerome Salinger	jeromes@yahoo.com	89269548736	

Now in case of change of the contractor's phone number "Alla Leonidovna Amelina", the following changes will be made in the table LOG_CHANGES and LOG_FIELDS_CHANGED *of the kernel scheme*, as well as table AGENTS *of the subject scheme* of the database:

AGENTS

ID	NAME	EMAIL	PHONE	LOG_ID
1	Ken Kesey	kenk@gmail.com	89262345124	
2	John Steinbeck	jhons@gmail.com	89031894536	
3	Kurt Vonnegut	kurtv@yahoo.com	89105238734	
4	Harper Lee	harperl@gmail.com	89031112233	142
5	Jerome Salinger	jeromes@yahoo.com	89269548736	

LOG_CHANGES

ID	TYPE	TIME	LOG_ID	TABLE_ID	U
245	u	2011.11.10 15:20:34	142	5	2

LOG_FIELDS_CHANGED

ID	TYPE	TIME	CHANGE_ID	FIELD_ID	NUMBER_VALUE	D
972	n	2011.11.10 15:20:34	245	19	89031112233	

Table *STAT_COMMAND_EVENTS* keeps information about the calls of handlers methods (filled in automatically):

- *START_DT* – call start or end time. Each call store in table *STAT_COMMAND_EVENTS* in two lines: The first line is the starting of call, the second - is the end. The rows can be differentiated by the values of *START_DT* and *DURATION* properties. It is the call start time for the first row *START_DT.DURATION* property has no such value. It is the call end time for the second row *START_DT.DURATION* property is assigned with the value;
- *DURATION* – call duration;
- *HANDLER_OBJ_ID (FK)* – a handler, which methods were called;
- *USER_ID (FK)* – a user, on which behalf the handler was called;
- *SESSION_ID (FK)* – a session, during which a call was performed;

- *SERVER_CALL_ID* – ID of the server call, which preserves constant value after login to the application server and till logout.

Logging operation

Logging is realized by means of a package *PACK_LOG*, which includes two procedures:

- *ENABLE_LOGGING* – includes logging for the specified table and its columns:

```
PROCEDURE ENABLE_LOGGING(
        vTABLE VARCHAR2,
        vCOLUMN_LIST VARCHAR2 := NULL,
        vSCHEMA_NAME VARCHAR2 := NULL);
```

  - *vTABLE* – table name which has to be logged;
  - *vCOLUMN_LIST* – complete list of columns names , divided by commas, which have to be logged. If *NULL*, then logging is enabled for all table columns except *LOG_ID*;
  - *vSCHEMA_NAME* – scheme name in which there is a specified table. If *NULL*, the scheme *kernel* kernel of database will be used.
- *DISABLE_LOGGING* – switches off logging for the specified table and its columns:

```
PROCEDURE DISABLE_LOGGING(
        vTABLE VARCHAR2,
        vCOLUMN_LIST VARCHAR2 := NULL,
        vSCHEMA_NAME VARCHAR2 := NULL);
```

  - *vTABLE* – table name which logging needs to be forbidden;
  - *vCOLUMN_LIST* – complete list of columns names, divided by commas, which have not to be logged. If *NULL*, then logging is switched off for all table columns;
  - *vSCHEMA_NAME* – scheme name in which there is a specified table. If *NULL*, the scheme *kernel* kernel of database will be used.
- *GET_PAUSE_MODE* – returns 1, if logging is suspended in the current session and 0, if logging is enabled:

```
FUNCTION GET_PAUSE_MODE RETURN NUMBER;
```

- *SET_PAUSE_MODE* – pauses or resumes logging for the current session:

```
PROCEDURE SET_PAUSE_MODE(vPAUSE_MODE NUMBER);
```

  - *vPAUSE_MODE* – if the parameter is equal to 1, logging will be suspended, if 0 — it will be renewed;

DDL operators can be carried out in the procedures `ALTER TABLE table_name ADD column_name`, at the same time there is transaction COMMIT. To make a call of these procedures in the main transaction (server calls) can be dangerous!

It is also possible to operate logging by means of the manager *IHistoryService* (from namespace *Ultima*):

```
HistoryService.EnableLogging("kernel", "table_name");

HistoryService.DisableLogging("kernel", "another_table_name", "column1", "column2");

/* Hard optimization ! Danger ! */
using (HistoryService.PauseLogging())
{
   CloneAllPrices();
}
```

Applications and modules

Client application architecture

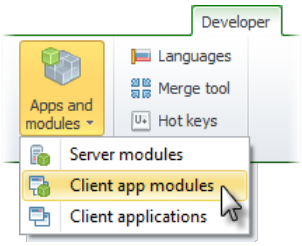
The client application is implemented in the form of a kernel and modules, which are loaded with the kernel from the application server. The composition of loaded modules is determined by the user role, under which the login to client application is carried out. The application server is implemented similarly in the form of a kernel and set of modules.

Moreover, the number of client applications can be more than one, as distinguished from the application server. In addition to client application for desktop PC, it can be for instance an application for touch screen PC or an application for mobile device.

In case of client application, division into modules is stipulated with existence of highly tailored functionality, which is not required for each user. For example, an ordinary user does not need a functionality of cash module or administrator, which in turn does not need the developer's functionality.

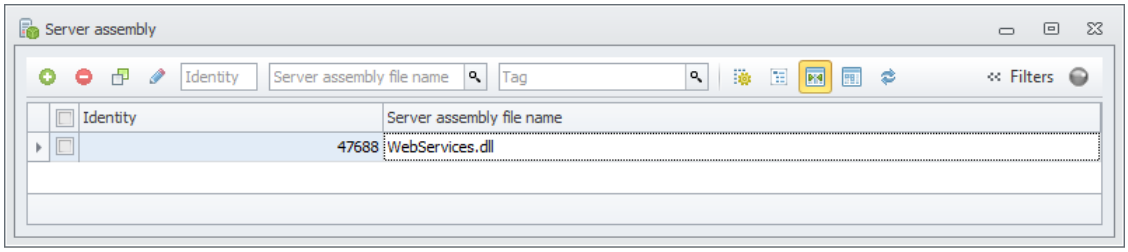
In turn, each module of the client application is a separate project, which includes a certain set of screen forms and commands. In terms of implementation, each module represents one and more libraries.

The list of all modules and applications can be found in the corresponding dictionaries:
Server modules – server modules;
Client app modules – modules of client applications;
Client application – client applications.



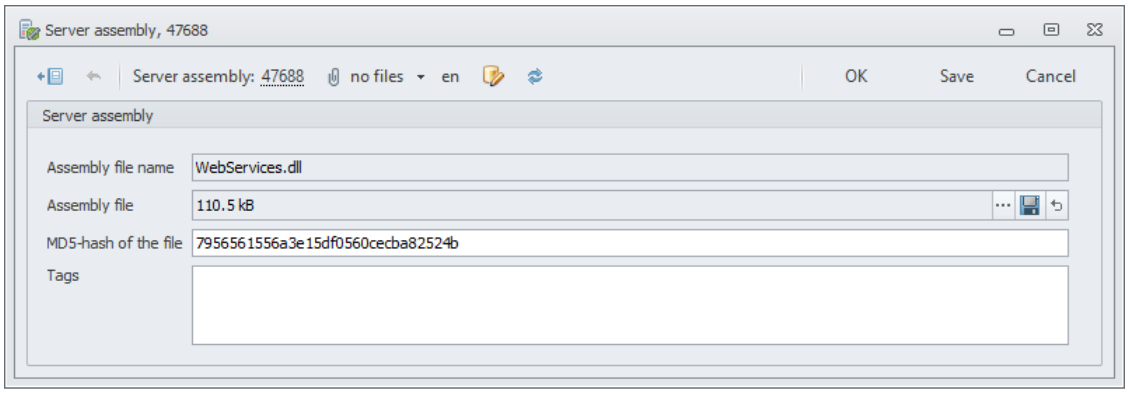
Server modules

The list of all server modules kept by an application server into its folder at start *ServerAssemblies*, it can be found in the dictionary “Server assembly":



Server modules can be filtered according to the *Module file name* (*Server assembly file name*) and *Tags* (*Tag*).

The server module edit form allows setting the following properties:



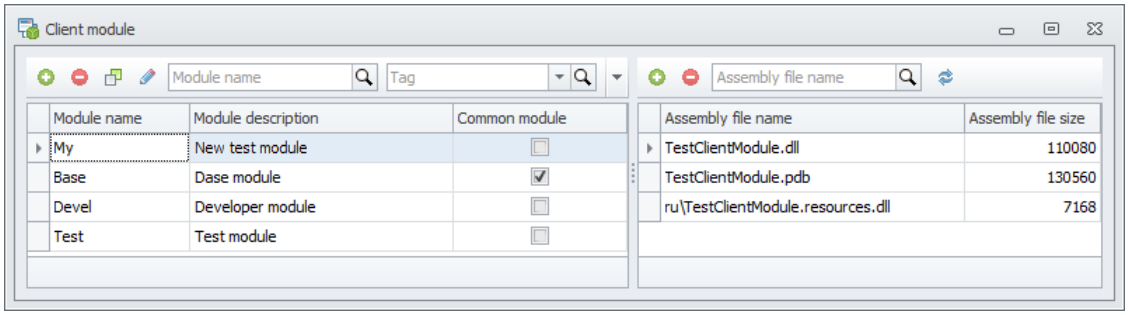
The following options are available to each attachment:

- *Assembly file name* – module file name, is put down automatically when selecting a file;
- *Assembly file* – module file name. Loading of dll and pdb-files is available (in order to line numbers were contained in exceptions of Stack trace). The size of the chosen file is displayed in the field:
  - button ··· opens the dialogue of loading of the module file;
  - button allows saving the copy of the file to the local disc of the computer or another available storage;
  - button opens the file. To open the program is used, associated with this file type in the operating system;
- *MD5-hash of the file* – MD5-hash of the module file;
- *Tags* – tags used for description of the module functionality. It is used to search the objects, realized under the certain functionality, associated with this tag.
  Addition of a tag is carried out by keys Space or Enter. Removal – button after the tag. As the space is used for input of the tag, it is possible to replace it with symbols "_" or "-" in tags with the name from several words.

In scripts it is possible to add references to assemblies of server modules.

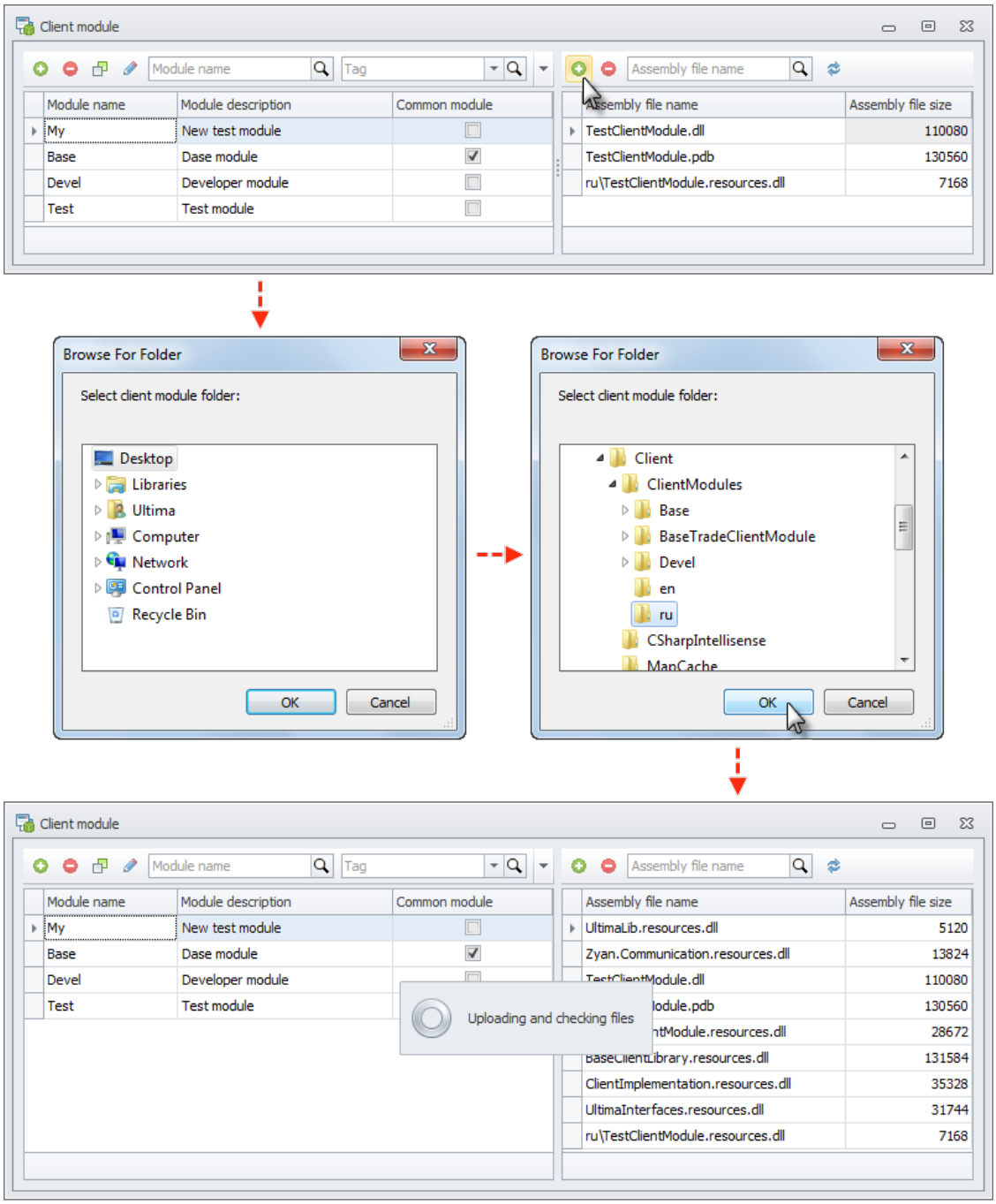
Modules of client applications

The list of all modules of client applications can be found in the dictionary "Client module":

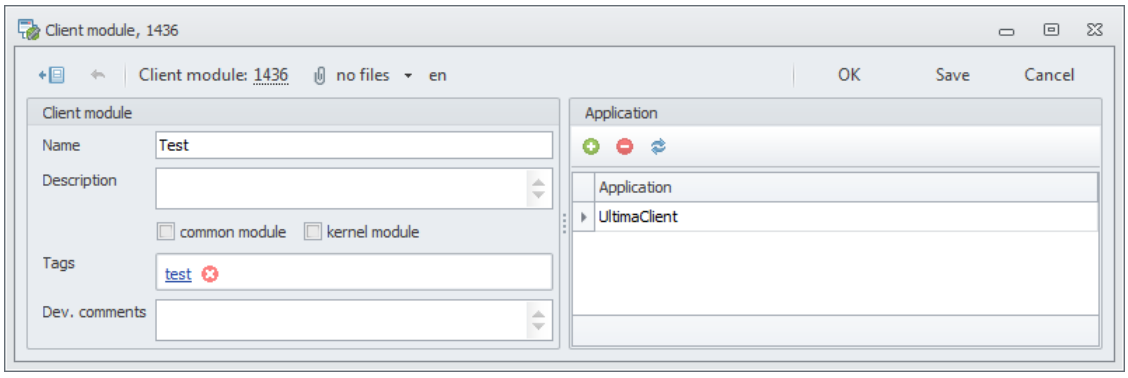


The dictionary window is divided into two parts: the modules of client applications are displayed on the left, the list of libraries (assemblies) of the module selected on the left is displayed on the right. The modules of client applications can be filtered by the *Module* name (*Name*) and *Tags* (*Tag*), their assemblies – by the *Name of the assembly file* (*Assembly file name*).

The assemblies for selected client module can be added or removed with corresponding buttons in the toolbar of assemblies. Addition is carried out by selection of the folder containing the assemblies. Moreover, all content of the folder (all files) will be added to selected module and loaded into the database:



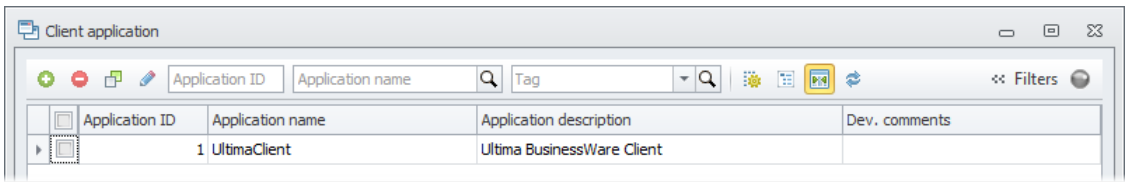
The edit form of the module allows setting the following properties:



- *Name* – module name. The name must coincide with the module folder name in the client application *Client**ClientModules****folder name**\;
- *Description* – module description;
- *common module* – a flag indicating that the module is common. For loading of common module, the user must not have any permissions. That is, while making the module common, we allow its loading by any user launching its application;
- *kernel module* – a flag indicating that the module is kernel. For loading of kernel module, the user must have corresponding permissions;
- *Tags* – tags used for description of the module functionality.
- *Dev. comments* – comments of application developer;
- *comments* – comments to the module. The comments are entered for each of system languages in the form opened by clicking the link;
- *Application* – a list of client applications, which the module is a part of. The applications can be added or removed with corresponding buttons in the toolbar. In case of adding, a list form will open for the client applications, where you can choose existing applications or create new ones. In case of removal, the application will be removed from the list of applications, which the module is a part of, but not from the dictionary of client applications.

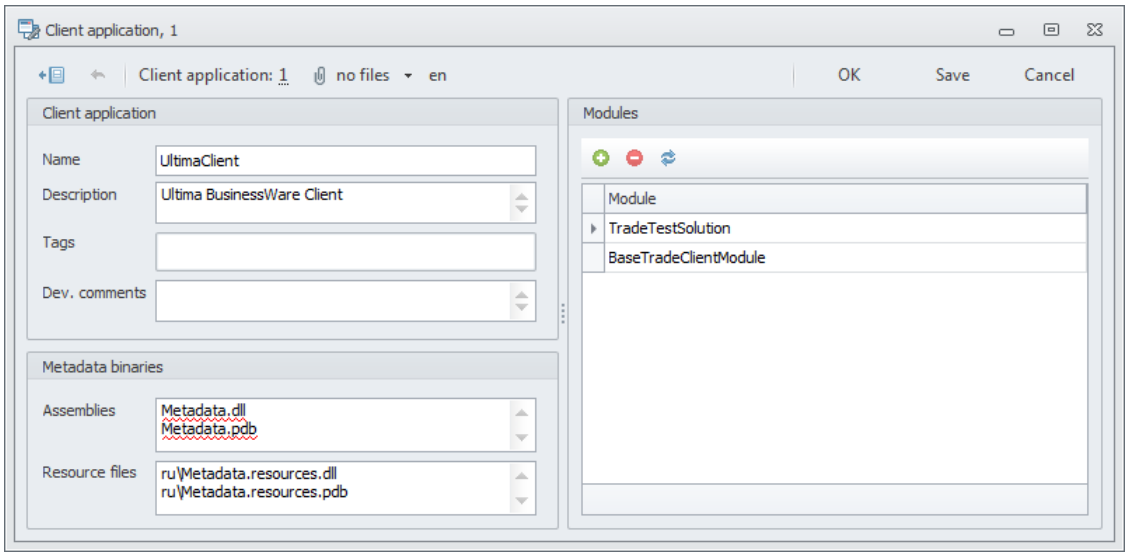
Client applications

The list of all client applications can be found in the dictionary "Client application":



The client applications can be filtered by *Name of the application* (*Application name*) and *Tags* (*Tag*).

The edit form of the client application allows setting the following properties:



- Group of properties*Client application* describes the client application:
  - *Name* – application name;
  - *Description* – application description;
  - *Tags* – tags used for description of the application functionality;
  - *Dev. comments* – comments of application developer;
- *Metadata Binaries* group of properties describes the metadata files, which the client application must load. In addition to the names of files, a path should be indicated to them in relation to the client application:
  - *Assemblies* – files of assemblies (libraries);
  - *Resource files* – files of resources;

  There are the following files of metadata assemblies:
  - *metadata.dll* – metadata;
  - *webservices.dll* – description of DTO for web services;
- *Modules* – a list of modules, which the client application includes. The modules can be added or removed using corresponding buttons in the toolbar. In case of adding, a list form will open for the modules of client applications, where you can choose existing modules or create new ones. In case of removal, the module will be removed from the list of modules, which comprise a part of client application, but not from the dictionary of client applications modules.

How to create modules and screen forms of main application_2

Each module of main client application is a separate project, which includes a certain set of *screen forms*, *commands* and *control elements*. The module must have at least one implementation of the class, derived from *BaseModule* class, if the commands of this module are assumed to be added to the interface (main menu).

DevExpress component library is used to create the screen forms. Every application developer must have a separate license for the set of components for *WinForms*. Other libraries are supplied as part of application and do not require additional licensing.

The platform of client application allows the application developer to create or override:
- modules of client application;
- list forms of dictionaries;
- edit forms of dictionary records;
- list forms of the documents (logs of the documents);

- edit forms of the documents;
- type of control elements of table parts;
- query forms for the parameters of interactive commands;
- user interface commands – these objects can be added to user interface menu and random logic can be implemented in them;
- random screen forms opened from the user interface commands;
- redefine the interface or replace the main form of the application in full.

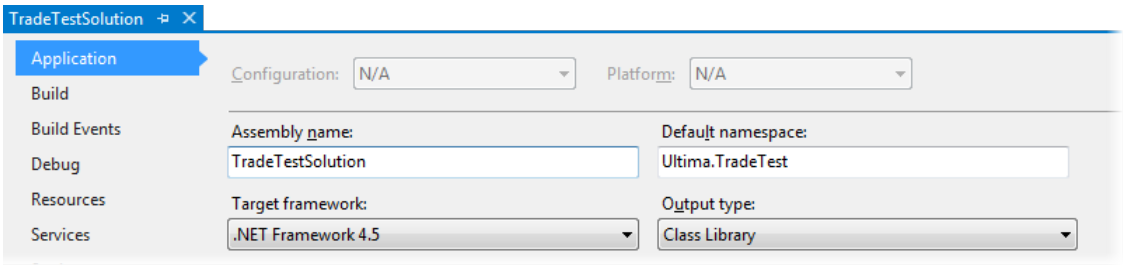
For each of the tasks, the application developer can use a set of classes, utilities and control elements provided by the application kernel.

Let us consider in due course which tools are available for solving each of the above tasks.

Modules

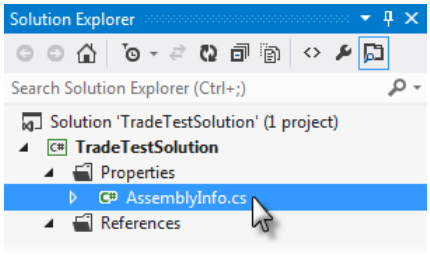
Let us consider how to create a new module stage by stage by the example.

Create a new project-library Class Library and set its properties:



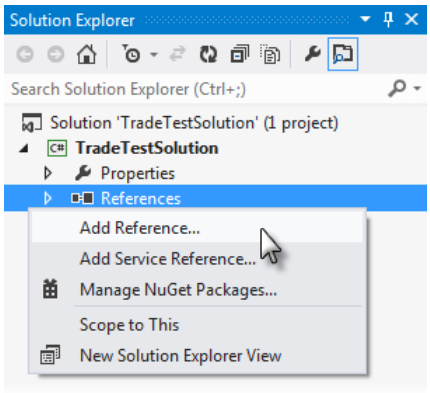
Edit the file *AssemblyInfo.cs*, while changing corresponding existing values for those given below:

```
AssemblyCompany(UltimaConstants.CompanyName)
AssemblyCopyright(UltimaConstants.Copyright)
AssemblyTrademark(UltimaConstants.Trademark)
AssemblyVersion(UltimaConstants.FullVersionString)
AssemblyFileVersion(UltimaConstants.FullVersionString)
```



Additionally connect the libraries:

- from the section *Assemblies -> Framework*:
  - System.ComponentModel.Composition;
  - System.Drawing;
  - System.WindowsForms;
- from the section Assemblies -> Extensions:
  - DevExpress.XtraGrid;
  - DevExpress.XtraEditors;
  - DevExpress.Data;
  - DevExpress.Utils;
- and additionally (by button Browse), indicating the files of the libraries from Ultimate AEGIS® distribution package:
  - BaseClientLibrary.dll – the library contains basic list forms and edit forms of the dictionaries and documents, basic form of the report, basic form of the query for parameters of interactive commands and control elements used in these forms;
  - ClientImplementation.dll – the library contains a class of basic module, used to produce a list of commands;
  - Metadata.dll – the library contains all classes of metadata;



- ClientInterfaces.dll – the library contains interfaces necessary also to create the commands and implementation of the list forms and edit forms;
- UltimaLib.dll – the library contains system constants and descriptor of the classes of metadata objects;
- UltimaInterfaces.dll – the library contains interfaces of special managers.

Create a folder *ThirdParty* in the project and add there a library from Ultimate AEGIS® distribution package with similar name;

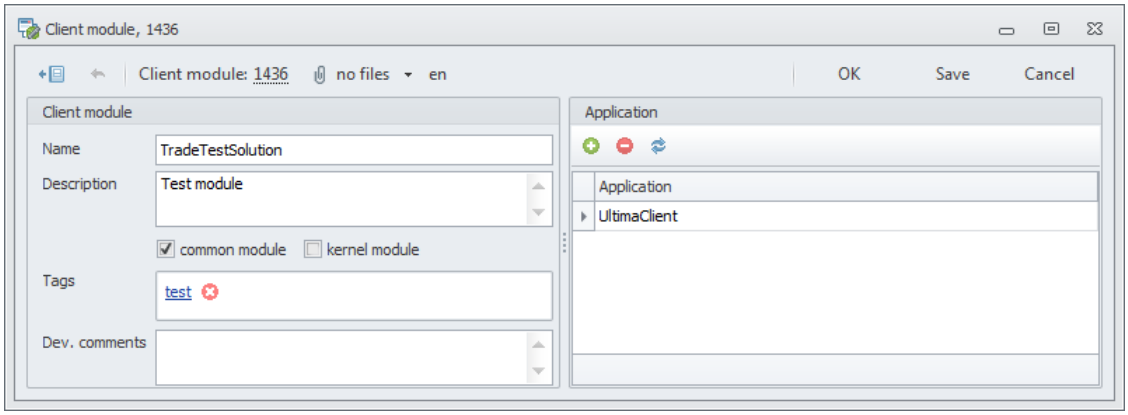
Connect *StyleCop*, intended for code analysis for compliance with the style:

- copy the file *Settings.StyleCop* from Ultimate AEGIS® distribution package into the project folder;
- edit the project file *.csproj, by adding a string:
  <Import Project="..\..\ThirdParty\StyleCop\StyleCop.targets" />

Creation of the module can be considered completed at this stage. Compile it and copy created libraries into the client application folder (it is the main client application of Ultimate AEGIS® in this example) Client:

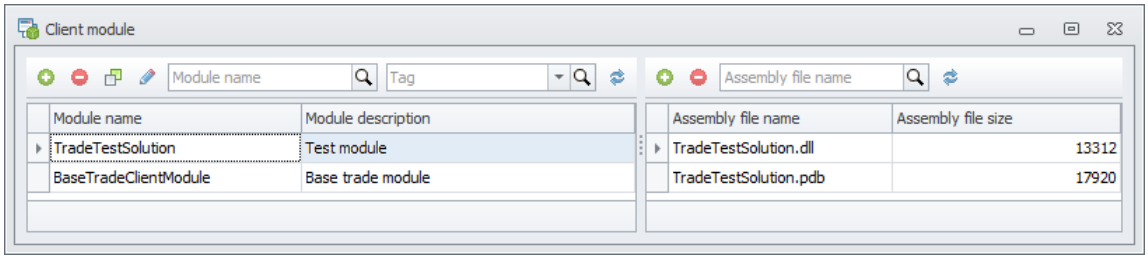
- *TradeTestSolution.dll* and *TradeTestSolution.pdb* into the folder *Client\ClientModules\TradeTestSolution*;
- *TradeTestSolution.resources.dll* into the folder *\Distrib\Client\ClientModules\TradeTestSolution\ru*.

The only remaining thing is to create corresponding module in the dictionary Client Module:



Module *Name* must coincide with the name of its folder in the client application *Client\ClientModules**TradeTestSolution***\.

Also add the created module into corresponding client application (UltimaClient), make it accessible (*common module*) and add the module libraries copied into the client application folder to it:



List forms of dictionaries

If the required level of the setting for interface of the list form of documents exceeds the possibilities offered by Ultimate AEGIS® system (see section System tools for setting of the interface of screen forms), own edit form can be created for any document.

The following hierarchy of classes is implemented in the system to facilitate the work:

BaseListForm
- *BaseDictionaryListForm*
  - *BaseFlatDictionaryListForm*
  - *BaseTreeDictionaryListForm*

BaseFlatDictionaryListForm is intended for implementation of the list forms of flat dictionaries, *BaseTreeDictionaryListForm* – tree-like ones.

To implement the dictionary list form, it should be derived from suitable form (flat or tree-like) and *IRecordBrowser<T>* and *IRecordSelector<T>* interfaces should be implemented, where *T* is type of dictionary.

The system will search for the form implementing *IRecordBrowser<T>* interface to display the list of records, and for selection of records (e.g. when clicked on ... in the control elements *DictionaryLookupEdit*) it will search for the form implementing *IRecordSelector<T>* interface. If no such form appears to be in the system, the basic dictionary list form will open. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator.

The application developer may request opening of the list form through *DictionaryHelper* using *SelectRecord<T>*, *SelectRecords<T>* or *BrowseRecords<T>* methods, where *T* is a type of dictionary, which form is required for opening. The first two methods describe the list form for selection of one or several records, the third method is used to view the list of records.

The most frequent reason to create own list form is implementation of the master-detail interface, which is also called a filter. Therefore, *BaseFlatDictionaryListForm* form hosts already *SplitContainer* control element, on which right panel a component is located for viewing of the dictionary records with all standard tools – selection of columns, filters, etc. The left panel is reserved for placement of the filter (table or tree), if this area is left empty – it will be hidden in the final form.

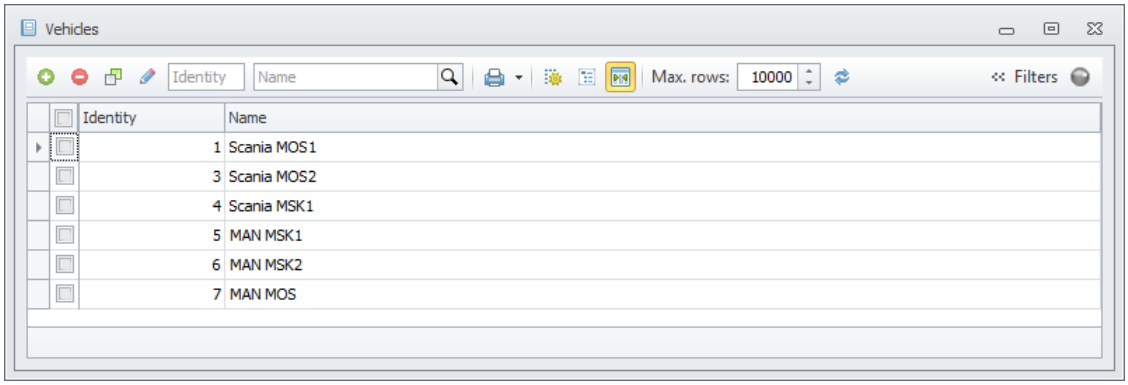
The application developer can also use the following control elements for creation of the list forms of dictionaries, e.g. for implementation of the filter:

- *DictionaryGridViewPanel* – to display the table with the toolbar;
- *DictionaryTreeViewPanel* – to display tree dictionaries with the toolbar.

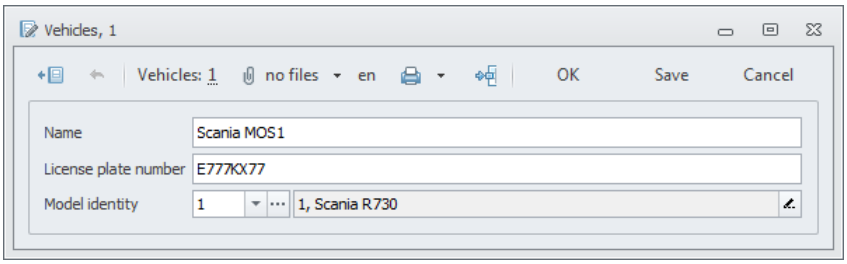
BaseTreeDictionaryListForm form is implemented in the same manner.

A full list of classes, forms and control elements can be viewed in the following section Ultima control elements.

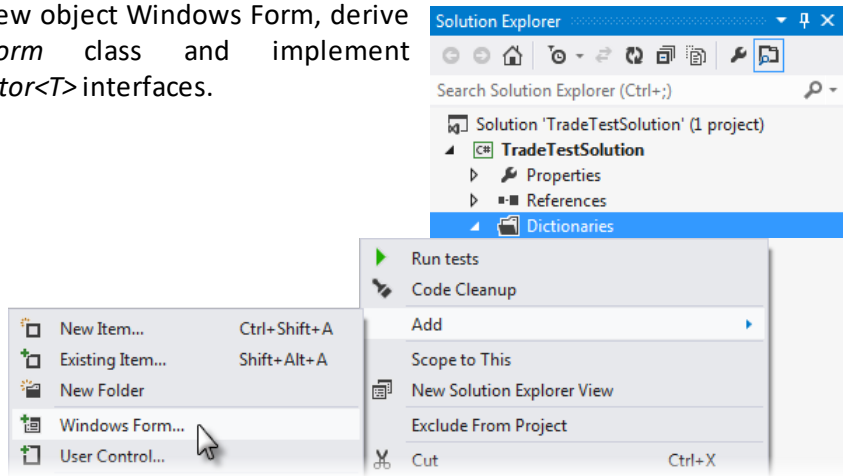
Let us consider creation of the list form by the example of the dictionary of vehicles Vehicle:



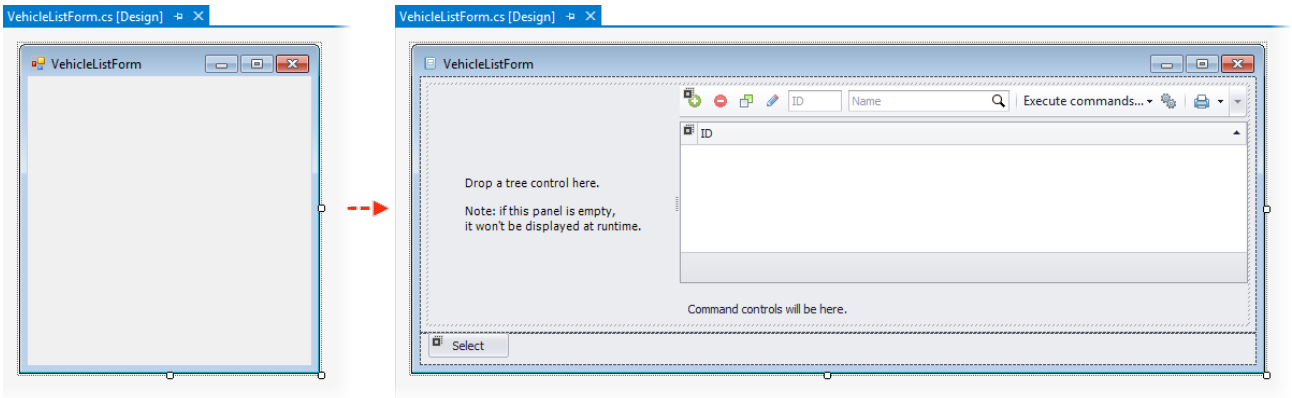
E.g. we will make so that the vehicles could be filtered by the selected model of the vehicle (records of the dictionary Vehicle model):



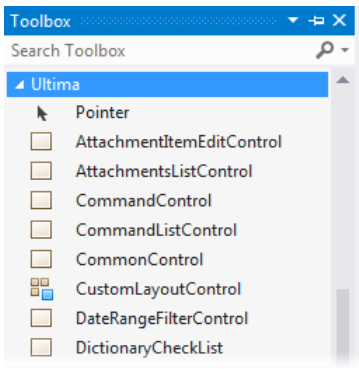
In the module project we create a new object Windows Form, derive it from *BaseFlatDictionaryListForm* class and implement *IRecordBrowser<T>* and *IRecordSelector<T>* interfaces.




```
public partial class VehicleListForm : BaseFlatDictionaryListForm,
        IRecordBrowser<Vehicle>, IRecordSelector<Vehicle>
{
        public VehicleListForm()
        {
                InitializeComponent();
        }
}
```

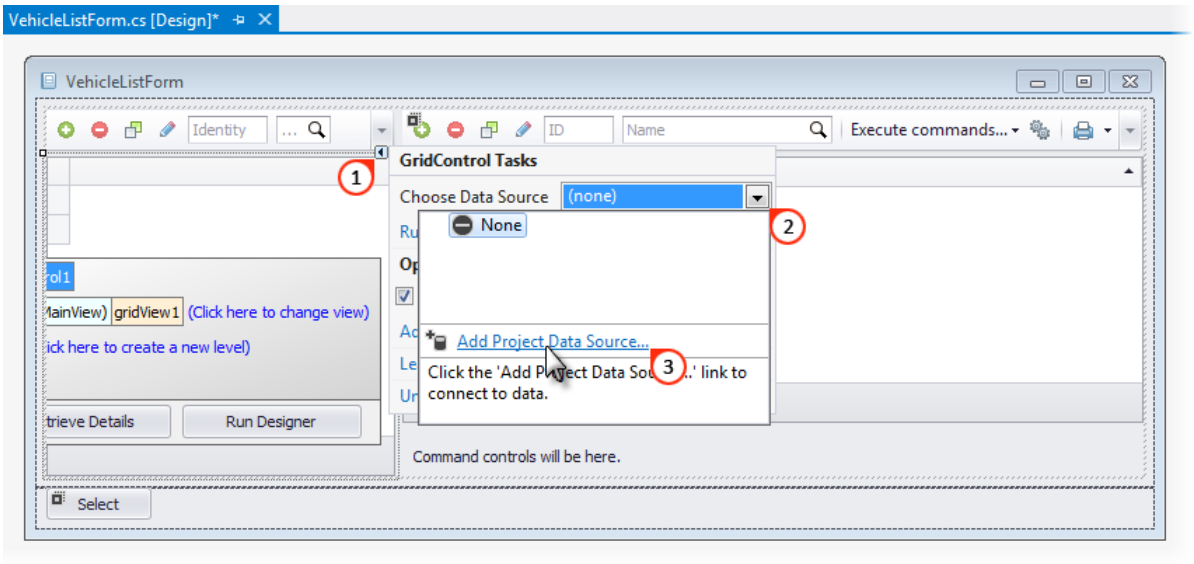


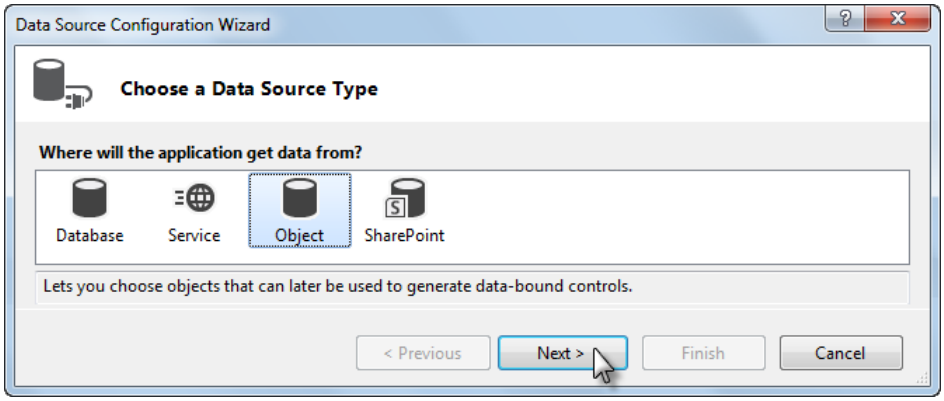
In order to get access to Ultima control elements, we have to create a folder (e.g. Ultima) in the Toolbox and add the control elements into it from the library BaseClientLibrary.dll (Choose Items -> Browse -> BaseClientLibrary.dll).

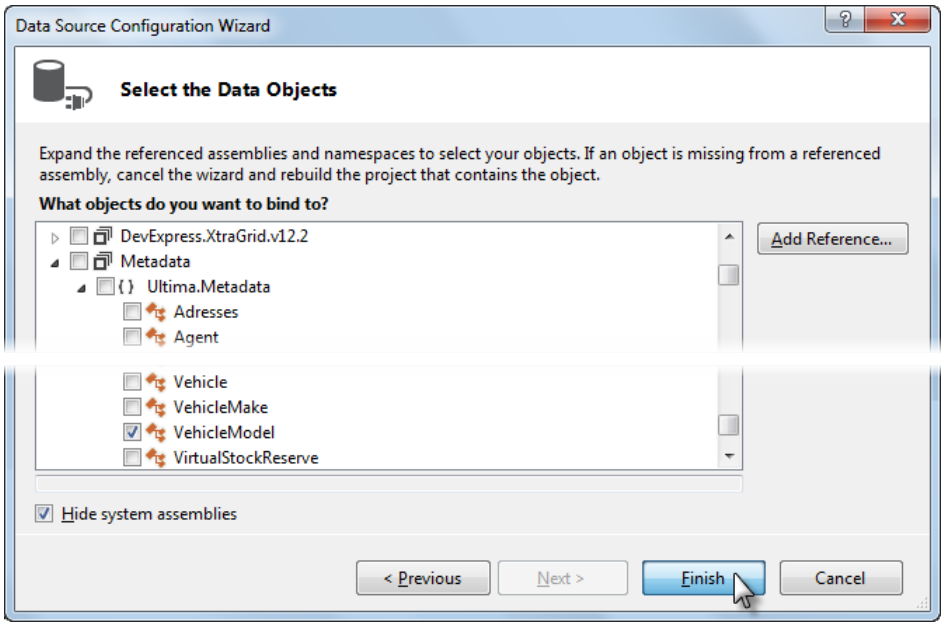


Use *DictionaryGridViewPanel* control elements to display the records of dictionary Vehicle model. Drag it onto the form and (in the properties) rename it for convenience into *VehicleModelGridViewPanel*. Choose the fill method *Dock* = *Fill*, in order to fill in the entire area of *SplitContainer* element.

Set the value *DictionaryType* = *Ultima.Metadata.VehicleModel* in the parameters to display a dictionary record in the control element. For its table (control element *GridControl* of the library DevExpress) and add similar data source:





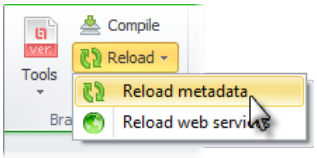


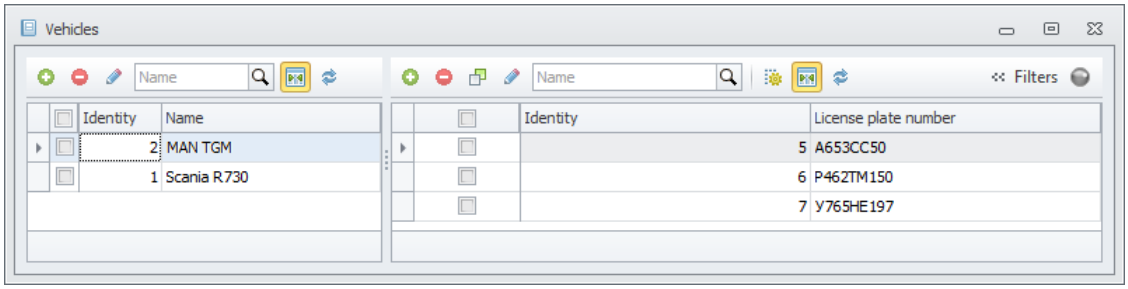
Implement necessary methods (which are described in the code):

```
public partial class VehicleListForm : BaseFlatDictionaryListForm,
       IRecordBrowser<Vehicle>, IRecordSelector<Vehicle>
{
       public VehicleListForm()
       {
              InitializeComponent();

              // In order the choice of the record of VehicleModel dictionary in
              // VehicleModelGridViewPanel element could influence on the content of
GridPanel, displaying
              // the records of the dictionary Vehicle, a filter should be added. We bind
              // the handler to the event ApplyCustomFilter of GridPanel control element.
              GridPanel.ApplyCustomFilter += GridPanel_ApplyCustomFilter;

              // In order in the created record of the dictionary Vehicle the model of the
vehicle
              // could be inserted automatically in accordance with the record of the
dictionary Vehicle model chosen
              // in VehicleModelGridViewPanel element,
              // a filter should be added. We bind the handler to the event
              // InsertRecord of GridPanel control element.
              GridPanel.InsertRecord += GridPanel_InsertRecord;

              // Remove odd tools from GridPanel.
              GridPanel.Properties.LimitCounterVisible = false;
              GridPanel.Properties.GroupButtonVisible = false;
              GridPanel.Properties.PrintButtonVisible = false;
              GridPanel.Properties.IDEditVisible = false;
              GridPanel.Properties.CommandsMenuVisible = false;
       }

       // Load the records into VehicleModelGridViewPanel control element.
       protected override async Task LoadRecords()
       {
              await VehicleModelGridViewPanel.LoadRecords();
              await base.LoadRecords();await base.LoadRecords();
       }
```

```
        // In case of change of model choice in \VehicleModelGridViewPanel element,
        // every time the list of records of GridPanel should be updated. For that purpose,
we will respond
        // to the event SelectionChanged of VehicleModelGridViewPanel element.
        private async void VehicleModelGridViewPanel_SelectionChanged(object sender,private
async void VehicleModelGridViewPanel_SelectionChanged(object sender,
                EventArgs e)
        {
                await GridPanel.LoadRecords();
        }

        // The handler of ApplyCustomFilter event of GridPanel element.
        private void GridPanel_ApplyCustomFilter(object sender,
                Client.Controls.CustomFilterEventArgs e)
        {
                var selectedModels = VehicleModelGridViewPanel.SelectedRecordList;
                var customFilter =
                        DictionaryFilterHelper.GetContainsFilterExpression("ModelID",
                        selectedModels, e.ParameterExpression);
                e.FilterExpressions.Add(customFilter);
        }

        // The handler of ApplyCustomFilter event of GridPanel element.
        private void GridPanel_InsertRecord(object sender, InsertRecordEventArgs e)
        {
                var selected = VehicleModelGridViewPanel.SelectedRecordList;
                if (!selected.IsEmpty)
                {
                        e.Parameters["ModelID"] = selected.First();
                }
        }
}
```

Compile a project, copy the created libraries into the module folder in the client application Client/ClientModules/TradeTestSolution, reload metadata and open the created list form of the dictionary using the same command, which was used for previous one:





How to create expression-subrequests in filters

LINQ expressions are used on the client for list and tree filtering. Simple predicates including constants or value lists are enough for filtering in most situations. For example, to display items out of the specified category list, that's enough to use this filter:

```
private void GridPanel_ApplyCustomFilter(object sender, CustomFilterEventArgs args)
{
	// filter by article groups not including subfolders
	var groups = ArticleGroupsTreePanel.SelectedRoots;
	args.AddFilter<Article>(ar => groups.Contains(ar.GroupID));
}
```

The problem appears when filter expression needs access to another table. Attempt to use *DataContext.GetTable()* doesn't lead to the expected results as LINQ-expressions on the client have restrictions connected to serialization and their transfer to the server. For example, this request will lead to a serialization error:

```
private void GridPanel_ApplyCustomFilter(object sender, CustomFilterEventArgs args)
{
	var groups = ArticleGroupsTreePanel.SelectedRoots;

	// filter by article groups including subfolders
	args.AddFilter<Article>(ar => groups.Contains(ar.GroupID) ||
		DataContext.GetTable<ArticleGroupTreeLink>()
			.Where(t => groups.Contains(t.AncestorID))
			.Select(t => t.DescendantID)
			.Contains(ar.GroupID));
}
```

It occurs because *DataContext* is form property in the expression. Actually it is *this.DataContext*, where *this* is a form which has a control element with a filter. Expression captures the link *this* as *Expression.Constant(typeof(FormType), this)*, and attempt of this expression transfer to the server fails as *FormType* is unavailable on the server.

To include the table reference in the filter, it is enough to replace *DataContext.GetTable* with the static *QueryExtensions.GetTable method* (it is necessary to connect *using Ultima.Data*). It is possible to use this method both in filters and in normal LINQ-requests, except for the very first table in the request. The filter shown in the previous example will take a form:

```
private void GridPanel_ApplyCustomFilter(object sender, CustomFilterEventArgs args)
{
	var groups = ArticleGroupsTreePanel.SelectedRoots;

	// filter by article groups including subfolders
	args.AddFilter<Article>(ar => groups.Contains(ar.GroupID) ||
		QueryExtensions.GetTable<ArticleGroupTreeLink>()
			.Where(t => groups.Contains(t.AncestorID))
			.Select(t => t.DescendantID)
			.Contains(ar.GroupID));
}
```

Such filters can be used not only for hierarchical tables. It is possible to filter, for example, the item list according to the list of online-categories. Online categories are connected to normal categories through the table *ArticleGroupsToOnline*:

```
private void GridPanel_ApplyCustomFilter(object sender, CustomFilterEventArgs args)
{
	// filter by online groups
	var onlineGroups = OnlineGroupsTreePanel.SelectedList;

	args.AddFilter<Article>(ar =>
		QueryExtensions.GetTable<ArticleGroupsToOnline>()
			.Where(link => onlineGroups.Contains(link.OnlineGroupID))
			.Select(link => link.ArticleGroupID)
			.Contains(ar.GroupID));
}
```

This method works only in the LINQ-requests executed on the client. In scripts it is still necessary to use *DataContext.GetTable()*.

Interceptor of the LINQ-requests executed from the client is added on the server. The interceptor processes LINQ-expressions, replacing one subtrees with others. In this case, having met *QueryExtensions.GetTable() method call*, the interceptor replaces it with *DataContext.GetTable() extension-method call*. As the interceptor processes only client requests, on the server *QueryExtensions.GetTable method* will lead to a broadcasting error in SQL.

How to create list form filters

If necessary, one can append additional properties and/or logic in addition to the standard ones, which can be selected via the columns selection form; the form is called by clicking the button F2).

However, this can be done only for the customized list forms, not for forms generated by the system by default. This restriction can be evaded once you create a custom copy of the list form in a few easy steps. E. g., for a flat dictionary, it is sufficient to create a new Windows Form object, inherit it from *BaseFlatDictionaryListForm* and implement *IRecordBrowser<T>* and *IRecordSelector<T>* interfaces, where *T* is a dictionary type (detailed information on the process provided on the initial stage of creation of Dictionary list form):

```
public partial class ArticleListForm : BaseFlatDictionaryListForm,
    IRecordBrowser<Article>, IRecordSelector<Article>
{
    public ArticleListForm()
    {
        InitializeComponent();
    }
}
```

There are two methods to add a property to the list form filter:

1. append one or several additional properties to the *User filter*. The properties appended this way will be originally available to all users. When configuring the *User filter* via the columns selection form called by clicking the button F2, the properties cannot be deleted;
2. configure a *Default filter* from the scratch by adding the desired properties into it and, if necessary, the standard ones (all of them or selectively).

Consider both methods using the example of the list form of the *Articles* dictionary. To put it simply, instead of a complicated logic, we will add to the filter the properties of the dictionary.

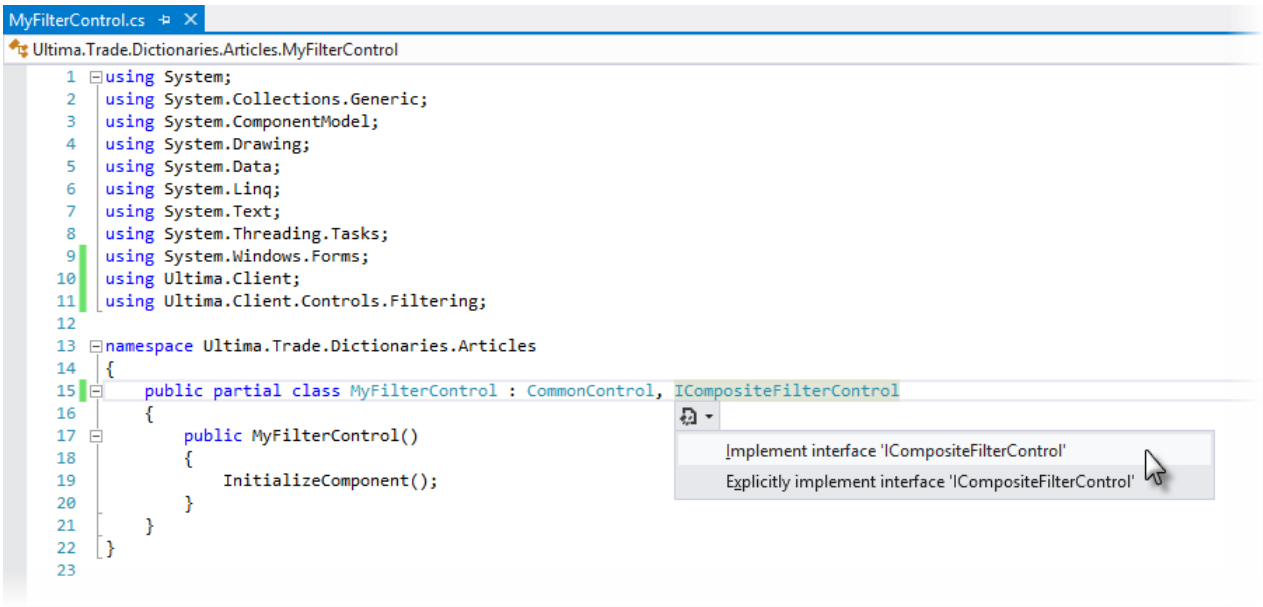
For the first example, let's add the property *Name* to the *User filter*. To do this, it is sufficient to implement in the list form class the following logic:

```
public ArticleListForm()
{
    InitializeComponent();

    AddAdditionalFilterControls();
}

// Controls for entering filter text
private TextEdit ArticleNameFilterEdit { get; set; }

private void AddAdditionalFilterControls()
{
    // Filter entry field caption
    var ArticleNameFilterLable = new LabelControl
    {
        Text = "Name",
        Dock = DockStyle.Top
    };

    // Control: filter entry field
    ArticleNameFilterEdit = new TextEdit
    {
        Name = "ArticleNameFilterEdit",
        Dock = DockStyle.Top
    };

    // Adding caption and filter entry field
    GridPanel.FilterControl.AdditionalFilterControls.Add(ArticleNameFilterLable);
    GridPanel.FilterControl.AdditionalFilterControls.Add(ArticleNameFilterEdit);
    ArticleNameFilterEdit.BringToFront();
    GridPanel.FilterControl.BestSizeAdditionalFilterControls();
}

// Applying filter
private void GridPanel_ApplyCustomFilter(object sender, CustomFilterEventArgs args)
{
    if (args.FilterActive && !string.IsNullOrWhiteSpace(ArticleNameFilterEdit.Text))
    {
        var name = ArticleNameFilterEdit.Text;

        args.AddFilter<Article>(a => a.name.ToLower().Contains(name));
    }
}
```

The second example implies the creation of a new control for the *Default filter*. Create a new element "User control", inherit it from *CommonControl* (from *Ultima.Client* namespace) and implement *ICompositeFilterControl* interface (from *Ultima.Client.Controls.Filtering* namespace):

```
public partial class MyFilterControl : CommonControl, ICompositeFilterControl
{
    public MyFilterControl()
    {
        InitializeComponent();
    }
}
```

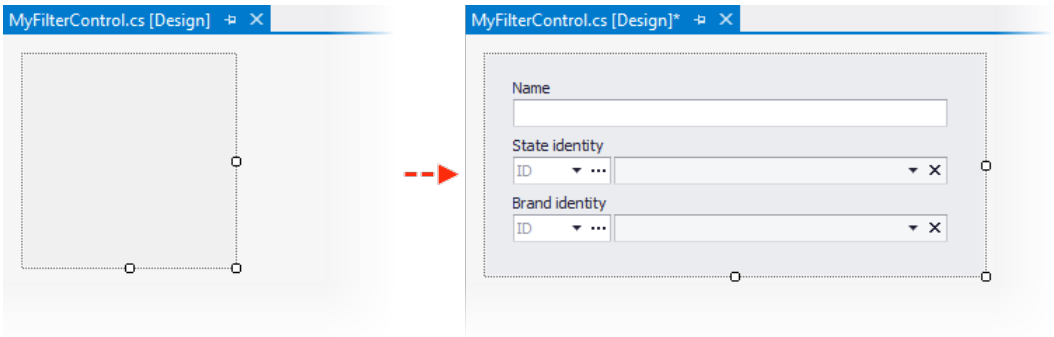
Implement *ICompositeFilterControl* interfaces:




```
public partial class MyFilterControl : CommonControl, ICompositeFilterControl
{
    public MyFilterControl()
    {
        InitializeComponent();
    }

    public event EventHandler ApplyFilters;

    public ClassDescriptors.IClassDescriptor ClassDescriptor
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    public event EventHandler FilterExpressionChanged;

    public Task<System.Linq.Expressions.LambdaExpression> GetFilterExpressionAsync()
    {
        throw new NotImplementedException();
    }

    public void ResetFilter()
    {
        throw new NotImplementedException();
    }

    public void ShowCustomizationForm()
    {
        throw new NotImplementedException();
    }
```

```
    public bool SupportsCustomization
    {
        get { throw new NotImplementedException(); }
    }
}
```

Provide the filter with the control elements for entering filter values: *Name*, *State* and *Brand*:



Implement filter logic:

```
public partial class MyFilterControl : CommonControl, ICompositeFilterControl
{
    public MyFilterControl()
    {
        InitializeComponent();
    }

    public event EventHandler ApplyFilters;

    // ApplyFilters event stub
    private void OnApplyFilters()
    {
        ApplyFilters.SafeInvoke(this, EventArgs.Empty);
    }

    public event EventHandler FilterExpressionChanged;

    // ApplyFilters event stub
    private void OnFilterExpressionChanged()
       {
              FilterExpressionChanged.SafeInvoke(this, EventArgs.Empty);
       }

    // Specify the dictionary in class descriptor
    public ClassDescriptors.IClassDescriptor ClassDescriptor
    {
              get { return Article.StaticClassDescriptor; }
              set { /* ignore */ }
       }

    // Applying filter
    public async Task<System.Linq.Expressions.LambdaExpression> GetFilterExpressionAsync()
    {
        // Filter object
        var filter = PredicateBuilder.Get<Article>();
```

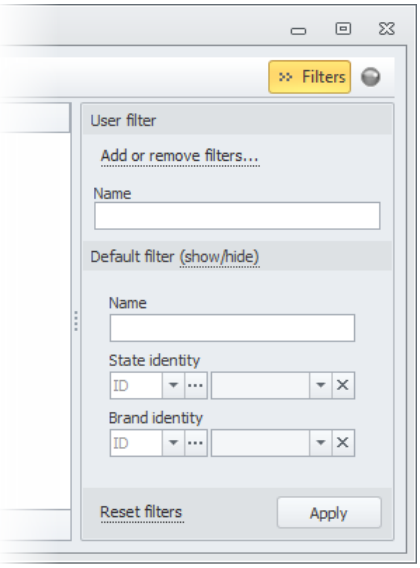
```
        // Filter conditions
        if (!string.IsNullOrWhiteSpace(NameEdit.Text))
        {
            var name = NameEdit.Text;
            filter = filter.And(a => a.Name.ToLower().Contains(name.ToLower()));
        }

        if (StateEdit.SelectedList.Any())
        {
            var stateList = StateEdit.SelectedList;
            filter = filter.And(a => stateList.Contains(a.StateID));
        }

        if (BrandEdit.SelectedList.Any())
        {
            var brandList = BrandEdit.SelectedList;
            filter = filter.And(a => brandList.Contains(a.BrandID));
        }

        return await Task.FromResult(filter);
    }

    // Reset filter values
    public void ResetFilter()
    {
        NameEdit.ResetText();
        StateEdit.ClearSelectedRecords();
        BrandEdit.ClearSelectedRecords();
    }

    public void ShowCustomizationForm()
    {
        throw new NotImplementedException();
    }

    // Disable filter setup by customization standard means
    public bool SupportsCustomization
    {
              get { return false; }
    }
}
```

In conclusion, specify the need to use the newly-created filter in the dictionary list form class:

```
public BaseArticleListForm()
{
    InitializeComponent();

    GridPanel.DefaultFilterControl = new MyFilterControl();
}
```

Upon completion, compile the project, copy the created libraries into the module folder in the client application *Client/ClientModules/TradeTestSolution*, reload metadata and open the filter of the *Articles* dictionary list form:

- *Name* field appended to the *User filter* by the first method is displayed in the upper part of the filter
- the fields appended to the *Default filter* by the second method are displayed in the filter's lower part.



Edit forms of dictionary records

If the required level of the setting for interface of the edit form of dictionary records exceeds the possibilities offered by Ultimate AEGIS® system (see section System tools for setting of the interface of screen forms), own edit form can be created for any dictionary.

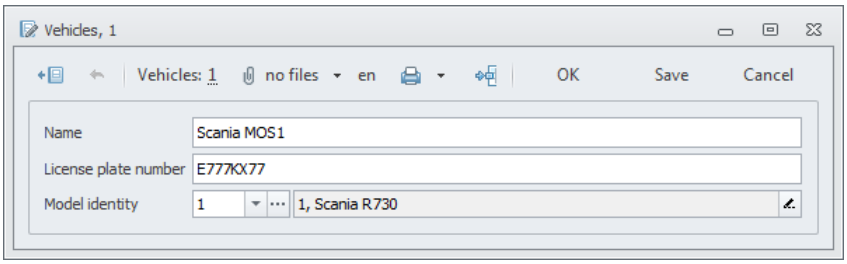
The following hierarchy of classes is implemented in the system to facilitate the work:
BaseEditForm
BaseDictionaryEditForm

To implement the edit form for dictionary records, it should be derived from *BaseDictionaryEditForm* form, and *IRecordEditor<T>* interface should be implemented, where *T* is a type of dictionary.

The system for editing of the dictionary record will search for the form implementing *IRecordEditor<T>* interface. If no such form appears to be in the system, the basic dictionary record edit form will open. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behavior of the system in case of error in the system setting by the administrator.

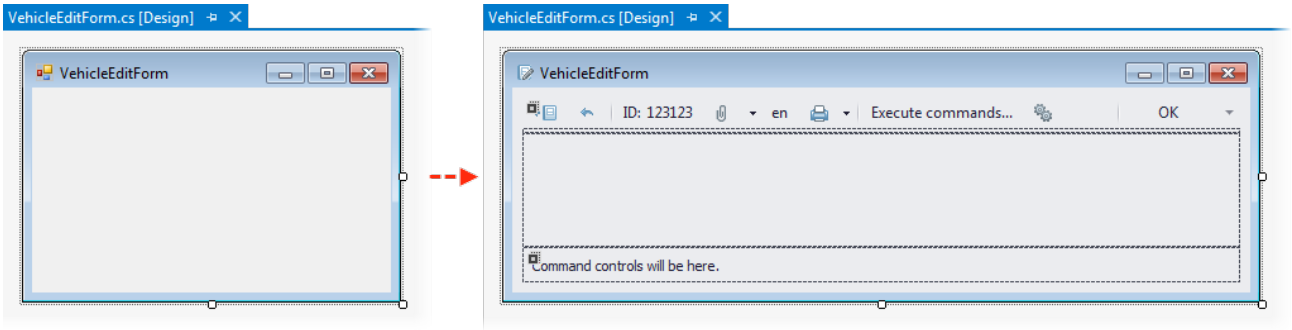
The application developer may request opening the dictionary record edit form through *DictionaryHelper* class using *EditRecord<T>* and *BeginEditRecord<T>* methods, where *T* is a type of dictionary, which record edit form should be opened. The methods open modal and modeless edit forms correspondingly.

Let us consider creation the edit forms for dictionary records by the example of the same dictionary of vehicles Vehicle:

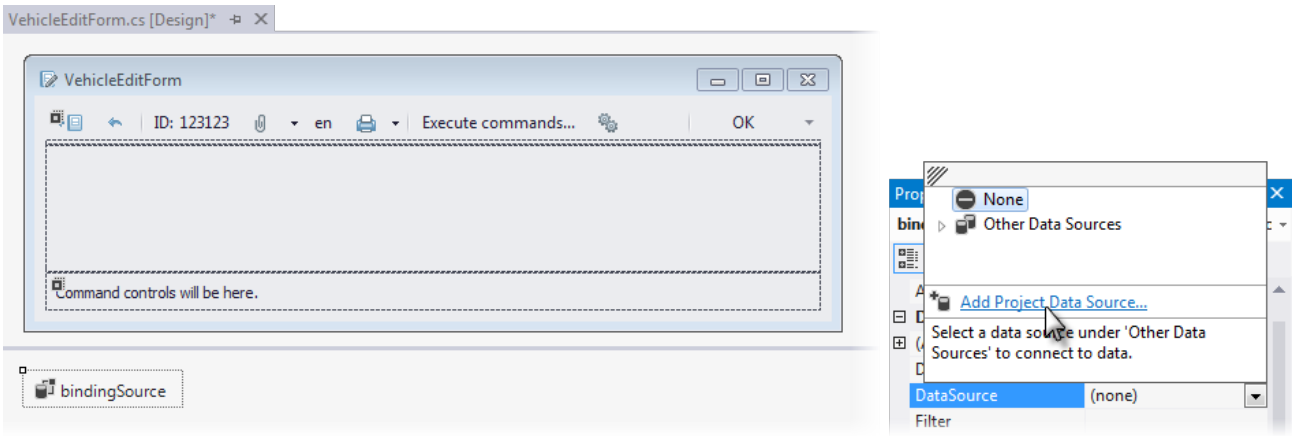


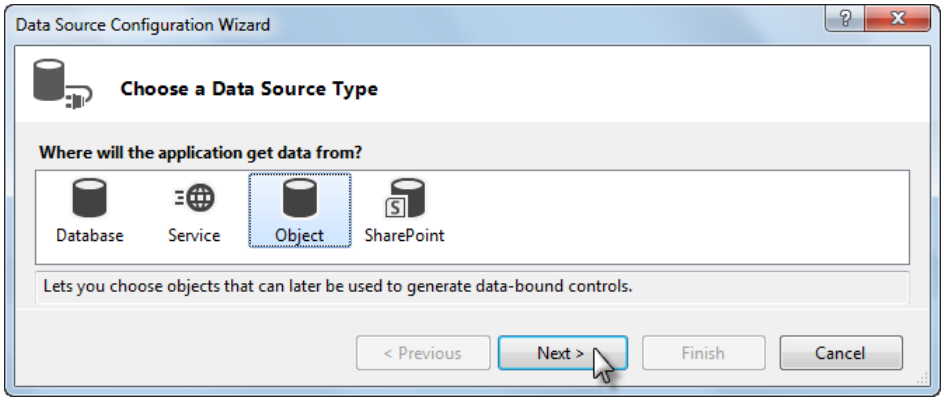
In the module project we create a new object Windows Form, derive it from *BaseDictionaryEditForm* class (from space name *Ultima.Client.Dictionaries*) and implement *IRecordEditor<T>*interface:

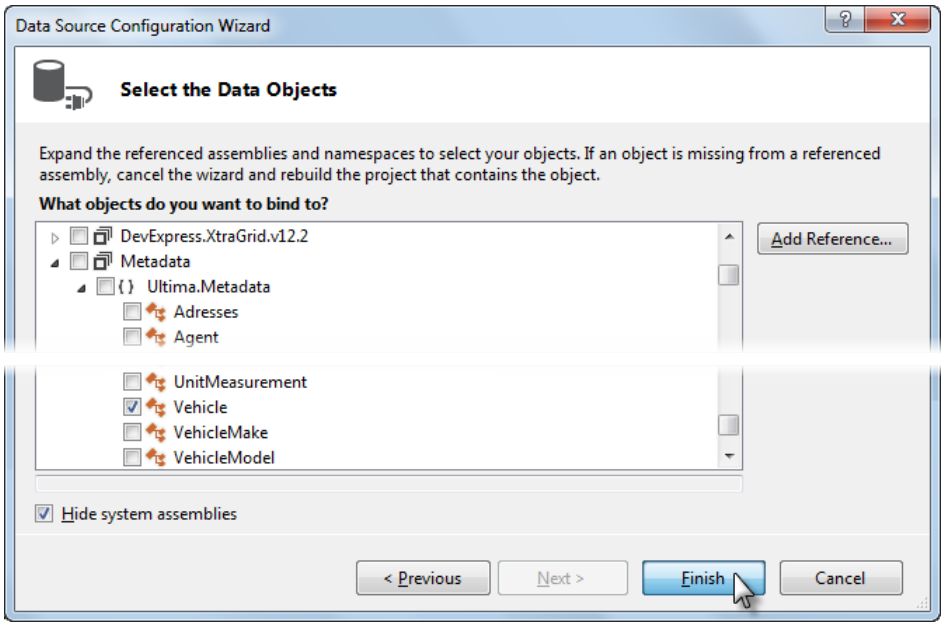
```
public partial class VehicleEditForm : BaseDictionaryEditForm, IRecordEditor<Vehicle>
{
	public VehicleEditForm()
	{
		InitializeComponent();
	}
}
```



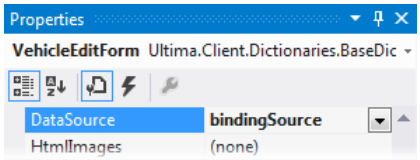
The data source should be connected to created form. For that purpose, add bindingSource control element to it and connect *Ultima.Metadata.Vehicle* metadata object to it:



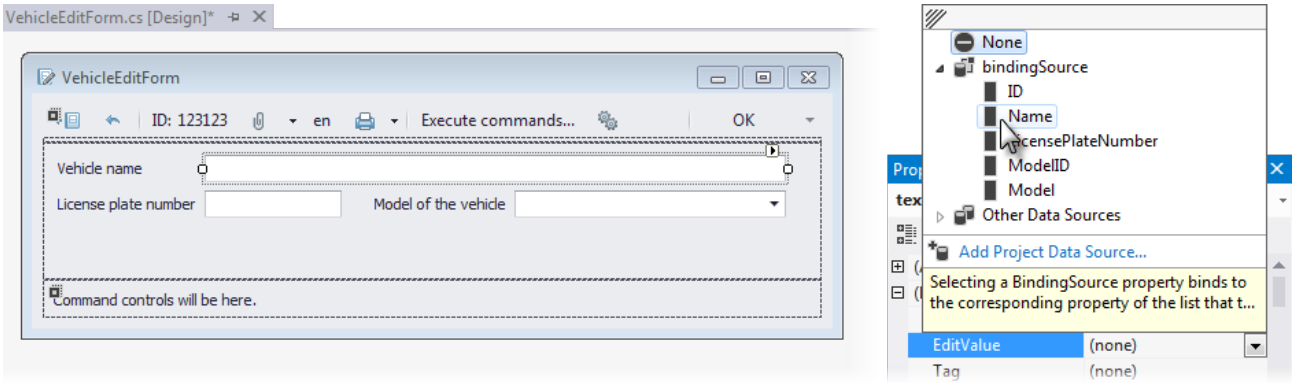




Assign the added bindingSource element in the form parameters as *DataSourse*:

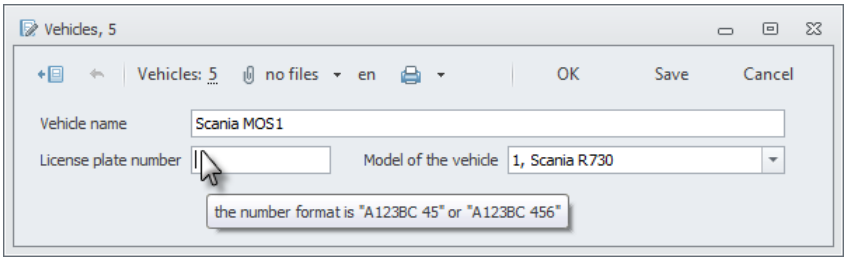


Now add the control elements to the form, which are intended to display the properties of dictionary record. Connect each of them through DataBindings -> EditValue property to corresponding properties of the dictionary in bindingSorce:



For [DictionaryLookupEdit]() control element, using which the vehicle model (*ModelID*) is displayed, we choose additionally the type of dictionary in DictionaryType property, which records it displays – *Ultima.Metadata.VehicleModel*.

Upon completion compile a project, copy the created libraries into the module folder in the client application *Client/ClientModules/TradeTestSolution*, reload metadata and open the created dictionary record edit form:



List forms of documents

If the required level of the setting for interface of the edit form of documents exceeds the possibilities offered by Ultimate AEGIS® system (see section System tools for setting of the interface of screen forms), own edit form can be created for any document.

The following hierarchy of classes is implemented in the system to facilitate the work:
BaseEditForm
BaseDocumentEditForm

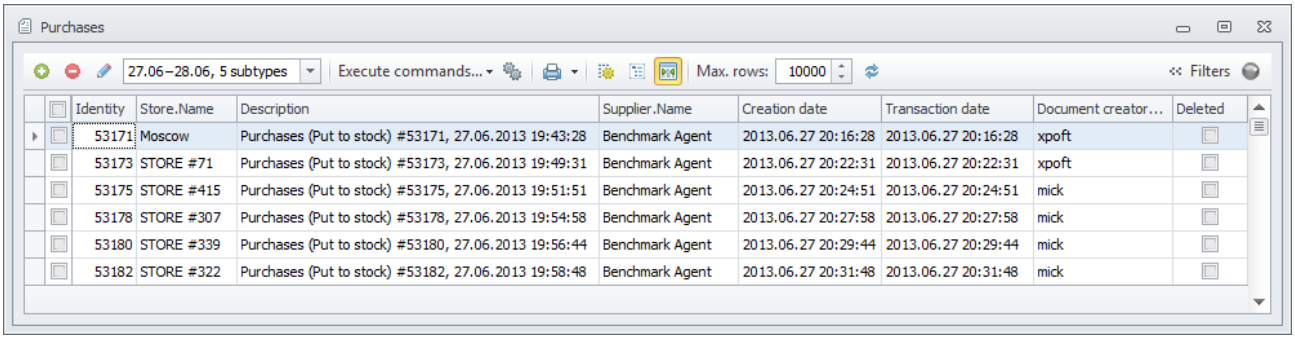
To implement the edit form for a document, it should be derived from *BaseDocumentEditForm* form, and *IRecordEditor<T>* interface should be implemented, where *T* is a document type.

The system for editing of the document will search for the form implementing *IRecordEditor<T>* interface. If no such form appears to be in the system, the basic document edit form opens. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator.

The application developer may request opening of the document edit form through *DocumentHelper* class using *EditDocument<T>* and *BeginEditDocument<T>* methods, where *T* is a type of document, which edit form should be opened. The first method opens the list form for selecting one document, the second one is used to view the list of documents.

The most common reason to create your own list form - master-implementation of interface details. Therefore, in the form BaseFlatDocumentListForm SplitContainer control is already posted , and on the right panel of it a component for the submission of documents with all standard tools - selection of columns, filters, etc. - is located. The left panel is reserved for the placement of a filter; if you leave this field empty it will be hidden in the final form.

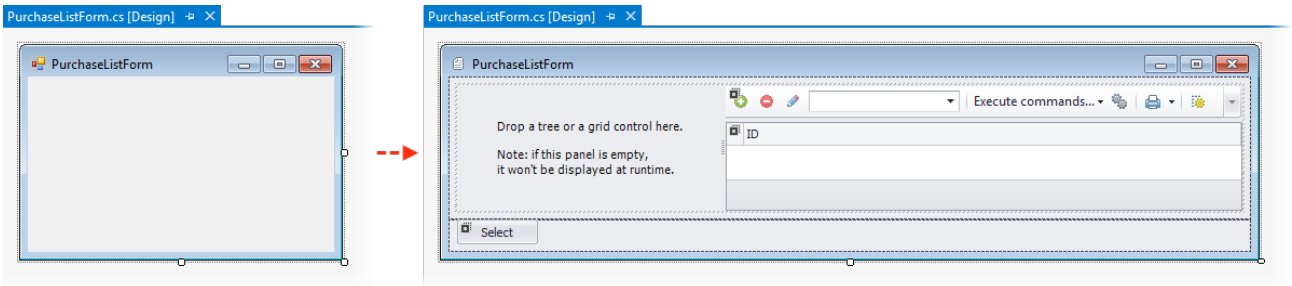
Let us consider creation of the edit form of documents by the example of Purchase type of documents:



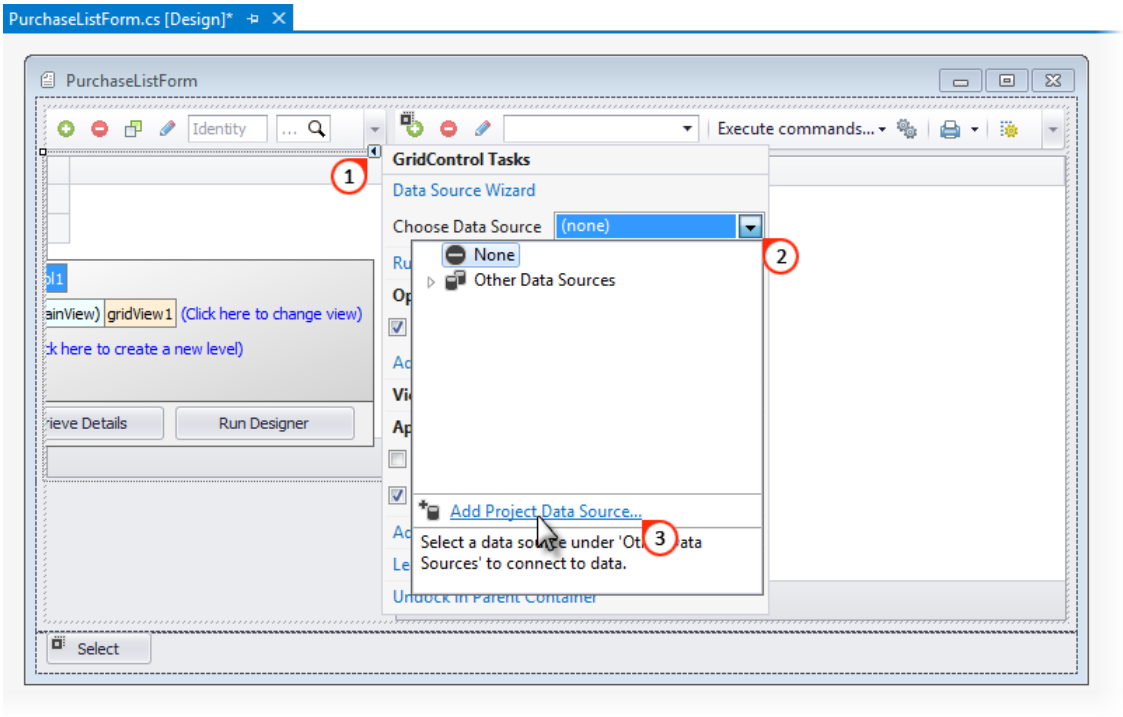
Identity	Store.Name	Description	Supplier.Name	Creation date	Transaction date	Document creator...	Deleted
53171	Moscow	Purchases (Put to stock) #53171, 27.06.2013 19:43:28	Benchmark Agent	2013.06.27 20:16:28	2013.06.27 20:16:28	xpoft	
53173	STORE #71	Purchases (Put to stock) #53173, 27.06.2013 19:49:31	Benchmark Agent	2013.06.27 20:22:31	2013.06.27 20:22:31	xpoft	
53175	STORE #415	Purchases (Put to stock) #53175, 27.06.2013 19:51:51	Benchmark Agent	2013.06.27 20:24:51	2013.06.27 20:24:51	mick	
53178	STORE #307	Purchases (Put to stock) #53178, 27.06.2013 19:54:58	Benchmark Agent	2013.06.27 20:27:58	2013.06.27 20:27:58	mick	
53180	STORE #339	Purchases (Put to stock) #53180, 27.06.2013 19:56:44	Benchmark Agent	2013.06.27 20:29:44	2013.06.27 20:29:44	mick	
53182	STORE #322	Purchases (Put to stock) #53182, 27.06.2013 19:58:48	Benchmark Agent	2013.06.27 20:31:48	2013.06.27 20:31:48	mick	

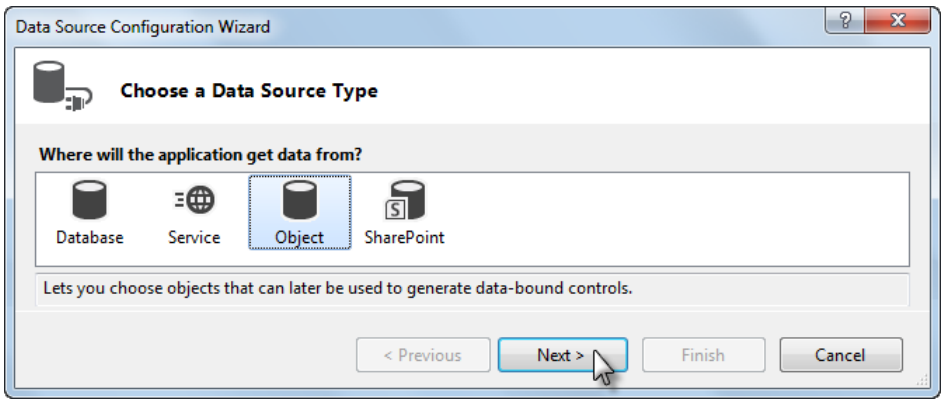
In the module project create a new object Windows Form, derive it from *BaseDocumentEditForm* (from *Ultima.Client.Documents namespace*) and implement *IRecordEditor<T>* interface:

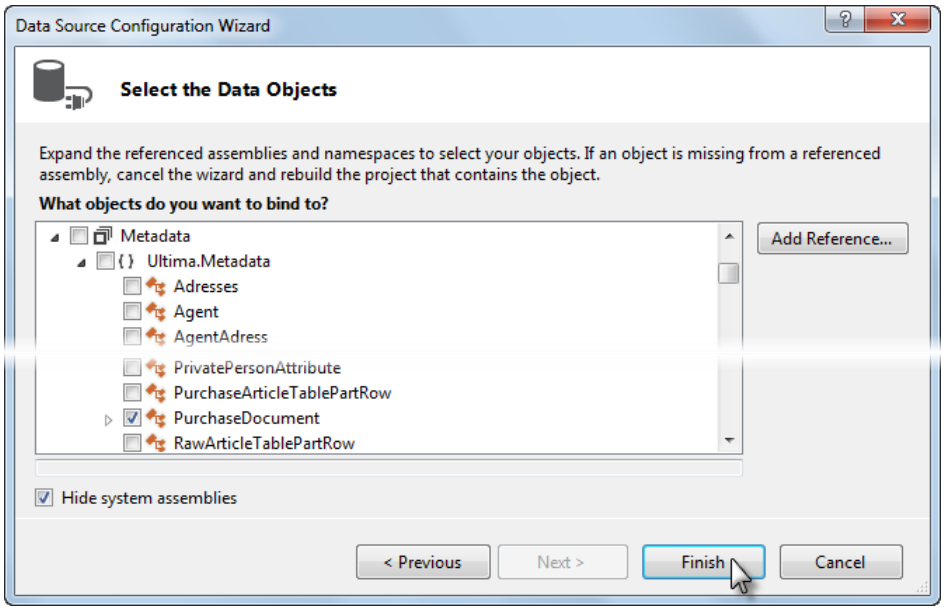
```
public partial class PurchaseEditForm : BaseDocumentEditForm,
IRecordEditor<PurchaseDocument>
{
	public PurchaseEditForm()
	{
		InitializeComponent();
	}
}
```



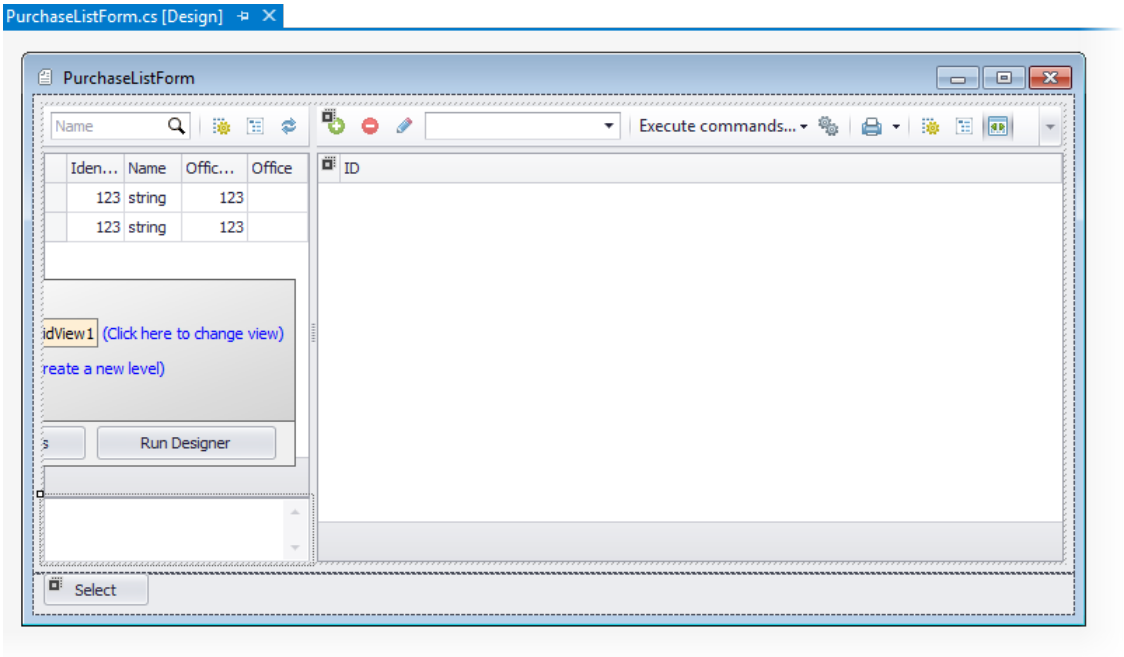
The data source should be connected to created form. For that purpose, add *bindingSource* control element to it and connect *Ultima.Metadata.PurchaseDocument* metadata object to it:







The control UltimaTextEdit is used to display the sub-type of the selected document.



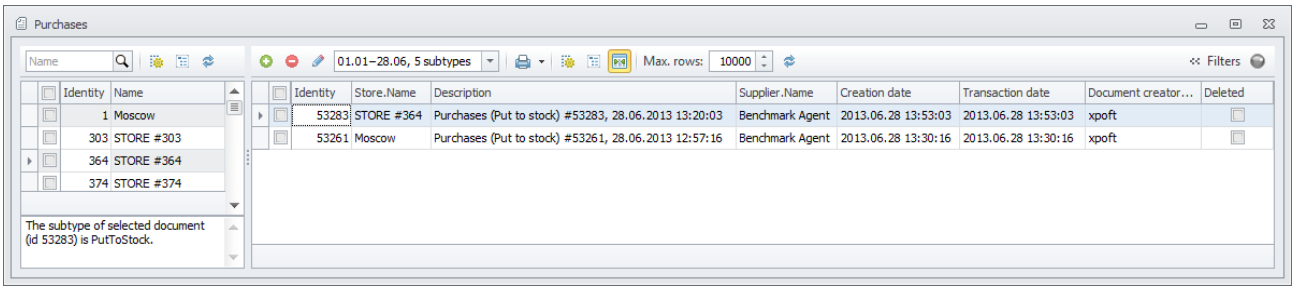
We realize the necessary methods (described in the code):

```
public partial class PurchaseListForm : BaseFlatDocumentListForm,
       IRecordBrowser<PurchaseDocument>, IRecordSelector<PurchaseDocument>
{
       // We import the manager, required to obtain the document subtype.
       [Import]
       private IDictionaryManager DictionaryManager { get; set; }

       public PurchaseListForm()
       {
              InitializeComponent();
```

```
            // If you want selection of Store Dictionary record in
            // StoreGridViewPanel element to influence GridPanel contents, showing
            // the documents, you should add a filter. Bind the handler
            // to ApplyCustomFilter event of GridPanel control.
            GridPanel.ApplyCustomFilter += GridPanel_ApplyCustomFilter;

            // If you want the store in the creating document
            // to be placed automatically in accordance with the Dictionary record Store,
            //  selected in the element StoreGridVeiwPanel, you should add a filter.
            // Bind the handler to InsertRecord event of GridPanel control.
            GridPanel.InsertRecord += GridPanel_InsertRecord;

            // For showing in the element UltimaTextEdit the subtype of document
            // selected in element GridPanel, you should also add a filter.
            // Bind the handler to SelectionChanged event of GridPanel control.
            GridPanel.SelectionChanged += GridPanel_SelectionChanged;
      }

      // Load entries in controls GridPanel and StoreGridViewPanel.
      protected async override Task LoadRecords()
      {
            await StoreGridViewPanel.LoadRecords();
            await base.LoadRecords();
      }

      // If you change a store in StoreGridViewPanel you should every time
      // upgrade the entries list in GridPanel. for this we will respond to
      // SelectionChanged event of  StoreGridViewPanel element.
      private async void StoreGridViewPanel_SelectionChanged(object sender, EventArgs e)
      {
            await GridPanel.LoadRecords();
      }

      // Handler ApplyCustomFilter of GridPanel element.
      private void GridPanel_ApplyCustomFilter(object sender,
            Client.Controls.CustomFilterEventArgs e)
      {
            var selectedStores = StoreGridViewPanel.SelectedRecordList;
            var customFilter =
                  DictionaryFilterHelper.GetContainsFilterExpression("StoreID",
                  selectedStores, e.ParameterExpression);
            e.FilterExpressions.Add(customFilter);
      }

      // Handler InsertRecord of GridPanel element.
      private void GridPanel_InsertRecord(object sender, InsertRecordEventArgs e)
      {
            var selected = StoreGridViewPanel.SelectedRecordList;
            if (!selected.IsEmpty)
            {
                  e.Parameters["StoreID"] = selected.First();
            }
      }

      // Handler SelectionChanged of GridPanel element.
      private void GridPanel_SelectionChanged(object sender, EventArgs e)
      {
            UpdateSummary();
      }
```

```
        // Show subtype of document selected in GridPanel in the element UltimaTextEdit.
        private void UpdateSummary()
        {
                var summaryText = string.Empty;
                if (SelectedList.Count == 0)
                {
                        summaryText = "No document selected.";
                        ultimaTextEdit1.Text = summaryText;
                        return;
                }

                if (SelectedList.Count > 1)
                {
                        summaryText = string.Format("Selected documents quantity {0}.",
                                SelectedList.Count);
                        ultimaTextEdit1.Text = summaryText;
                        return;
                }

                var documentId = SelectedID;
                var docSubtypeId = DocumentManager.GetDocumentSubtypeID(documentId);
                var docSubtypeName = DictionaryManager.
                        GetRecord<DocumentSubtype>(docSubtypeId).Name;
                summaryText = string.Format("Selected document subtype (id {0}) - {1}.",
                        documentId, docSubtypeName);
                ultimaTextEdit1.Text = summaryText;
        }
}
```

Upon completion we compile a project, copy the created libraries into the module folder in the client application *Client/ClientModules/TradeTestSolution*, reload metadata and open the created document edit form:



Edit forms of documents

If the required level of the setting for interface of the edit form of documents exceeds the possibilities offered by Ultimate AEGIS® system (see section System tools for setting of the interface of screen forms), own edit form can be created for any document.

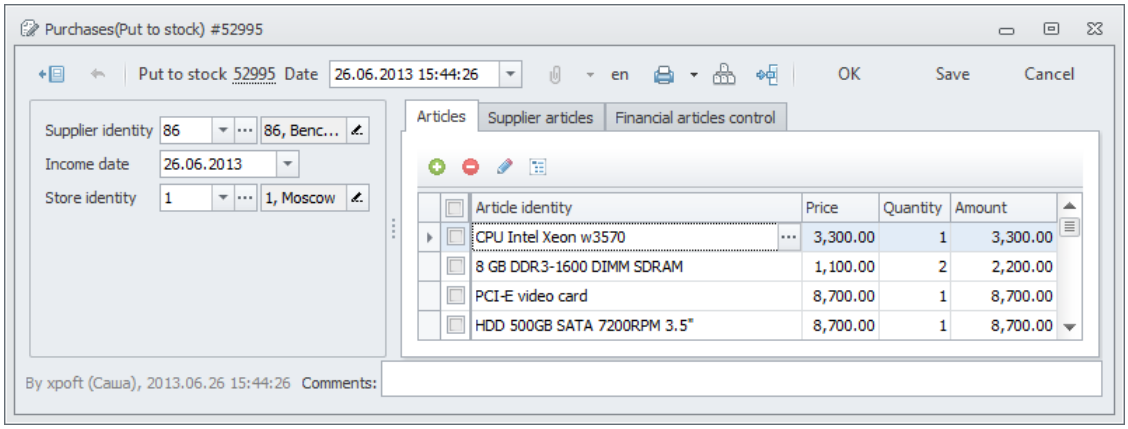
The following hierarchy of classes is implemented in the system to facilitate the work:
BaseEditForm
BaseDocumentEditForm

To implement the edit form for a document, it should be derived from *BaseDocumentEditForm* form, and *IRecordEditor<T>* interface should be implemented, where *T* is a document type.

The system for editing of the document will search for the form implementing *IRecordEditor<T>* interface. If no such form appears to be in the system, the basic document edit form opens. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator.

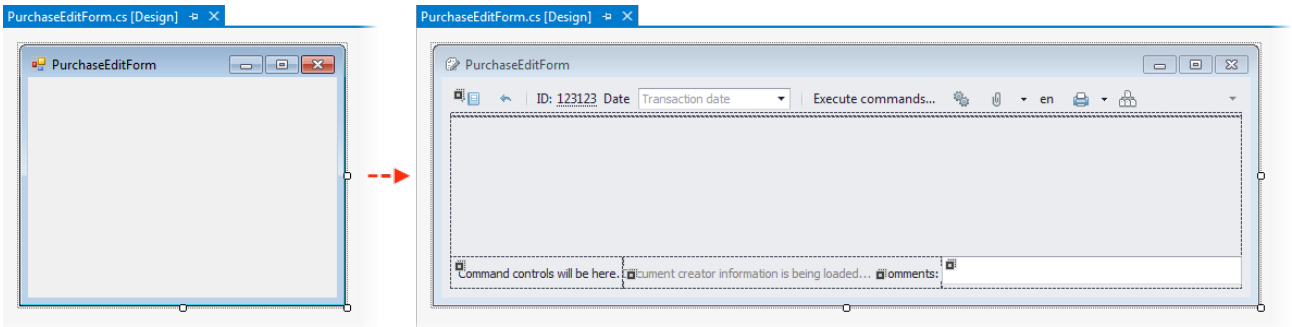
The application developer may request opening of the document edit form through *DocumentHelper* class using *EditDocument<T>* and *BeginEditDocument<T>* methods, where *T* is a type of document, which edit form should be opened. The methods open modal and modeless edit forms correspondingly.

Let us consider creation of the edit form of documents by the example of Purchase type of documents:

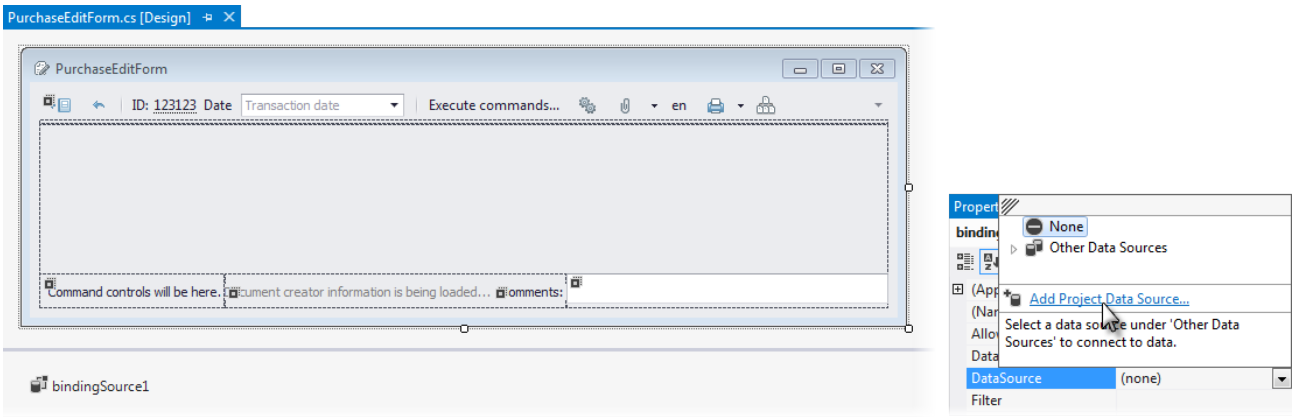


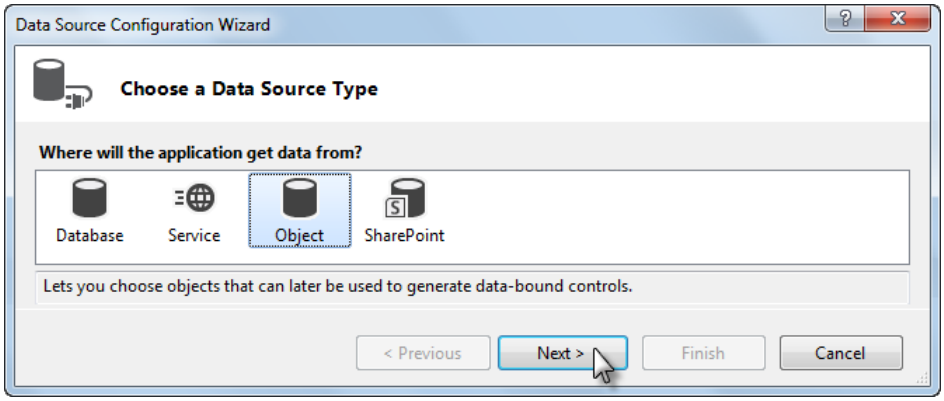
In the module project create a new object Windows Form, derive it from *BaseDocumentEditForm* (from *Ultima.Client.Documents namespace*) and implement *IRecordEditor<T>* interface:

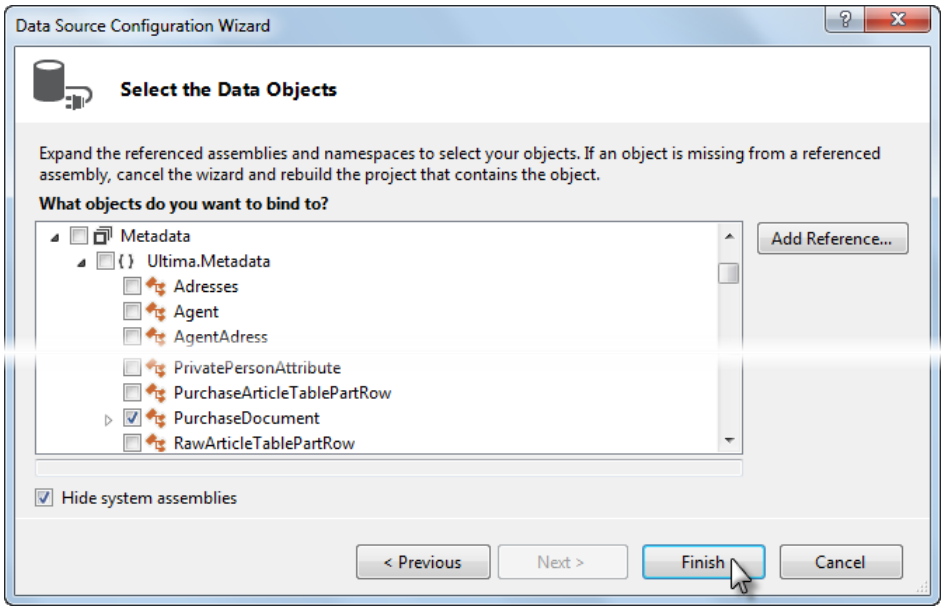
```
public partial class PurchaseEditForm : BaseDocumentEditForm,
IRecordEditor<PurchaseDocument>
{
	public PurchaseEditForm()
	{
		InitializeComponent();
	}
}
```



The data source should be connected to created form. For that purpose, add *bindingSource* control element to it and connect *Ultima.Metadata.PurchaseDocument* metadata object to it:

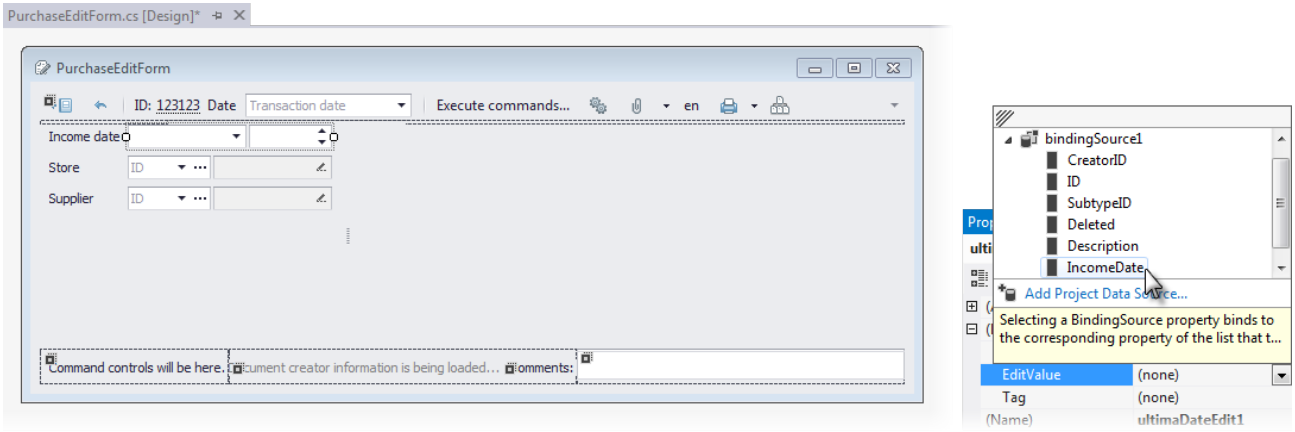






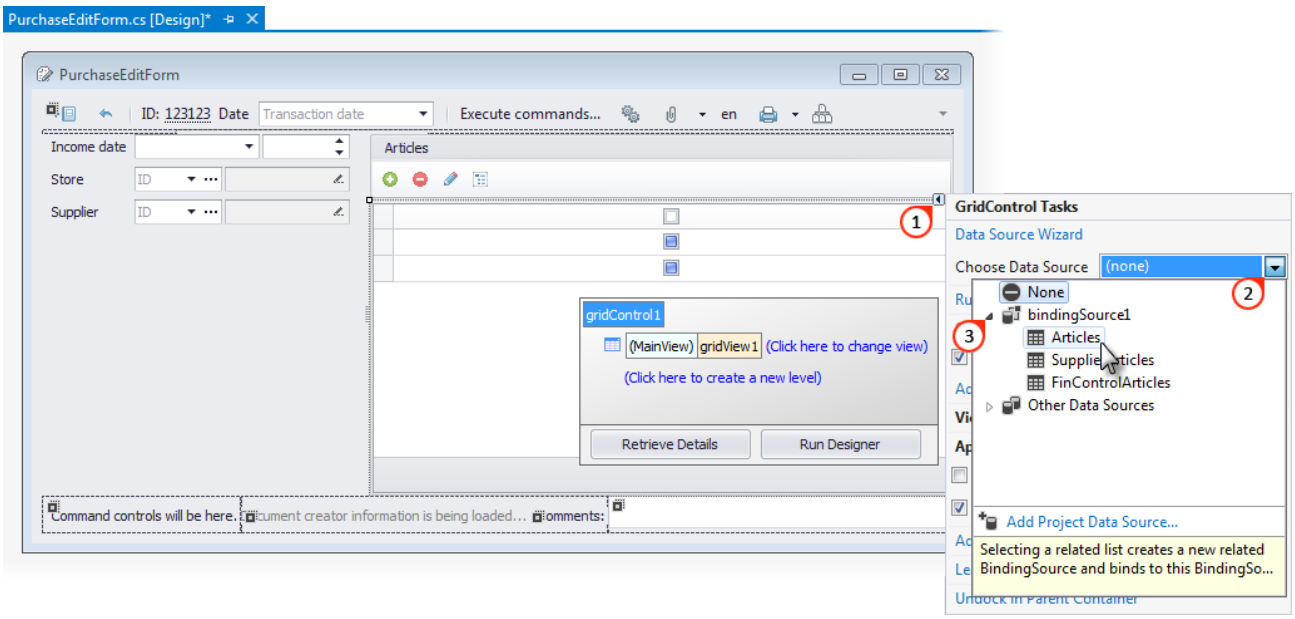
Assign the added *bindingSource* element in the form parameters as *DataSourse*.

Now we can add control elements to the form. In order to split the document properties and its table part, we use SplitContainerControl control element of DevExpress library. Each of the control elements, designed to display document properties, we connect through the property of DataBindings -> EditValue element to corresponding dictionary properties in bindingSorce:



For *DictionaryLookupEdit* control elements, using which the storehouse (*StoreID*) and supplier (*SupplierID*) is displayed, we additionally select a dictionary type in DictionaryType property, which records display – *Ultima.Metadata.Store* and *Ultima.Metadata.Agent* correspondingly.

In the left part of SplitContainerControl container, place *BaseTablePartGridPanel*, designed to display and edit the data of the document table part. Set a type of the table part for it through *TablePartType* parameter – *PurchaseArticleTablePartRow*. For its table (*GridControl* control element of DevExpresslibrary) connect the table part as data source, having selected it through previously added *bindingSource*:



Upon completion we compile a project, copy the created libraries into the module folder in the client application *Client/ClientModules/TradeTestSolution*, reload metadata and open the created document edit form:

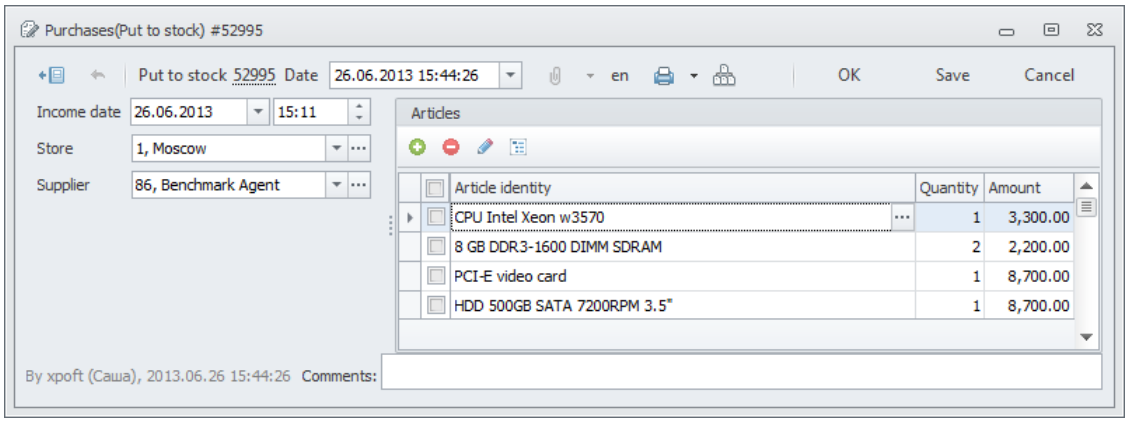


Table parts

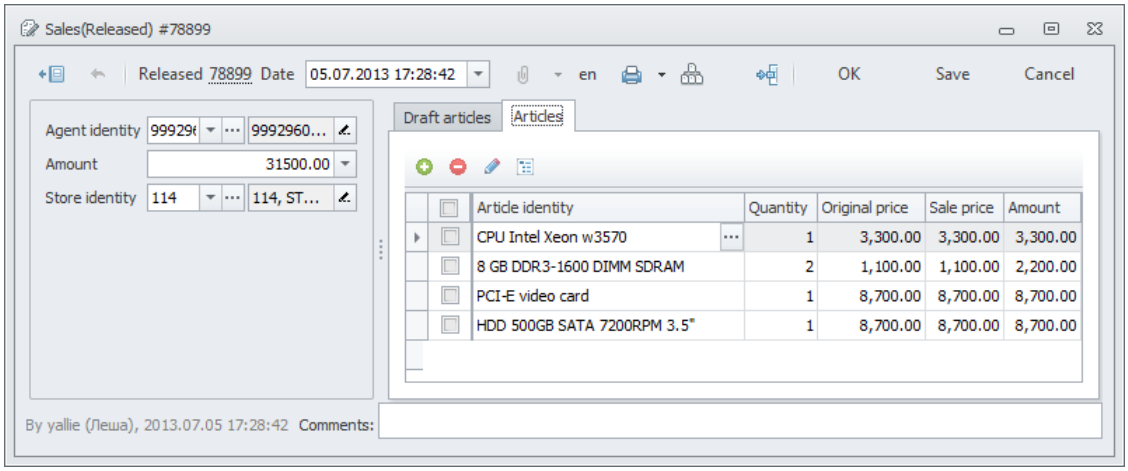
There is sometimes no need to change the entire document edit form but one of its table parts should be changed. In this case, own form can be created (control element) for any table part.

To implement the control element designed to edit of the table part, it should be derived from *BaseTablePartGridPanel* control element and *ITablePartEditor<T>* should be implemented, where *T* is a type of table part.

The system for editing of the document table part will search for the control element implementing *ITablePartEditor<T>* interface. If no such control element form appears to be in the system, the basic document table part edit form will open. If more than one of such control element appears to be in the system, the system will throw an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator.

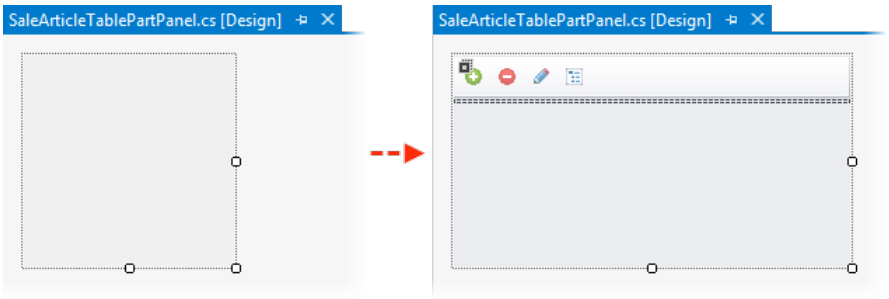
The application developer may request a control element for editing of the document table part through *DocumentHelper* class using *GetTablePartControl<T>* method, where *T* is a type of table part. The method returns a control element for editing of the table part of specified type.

Let us consider creation of the control element for editing of the document table part by the example of Article table part of document type Sale:



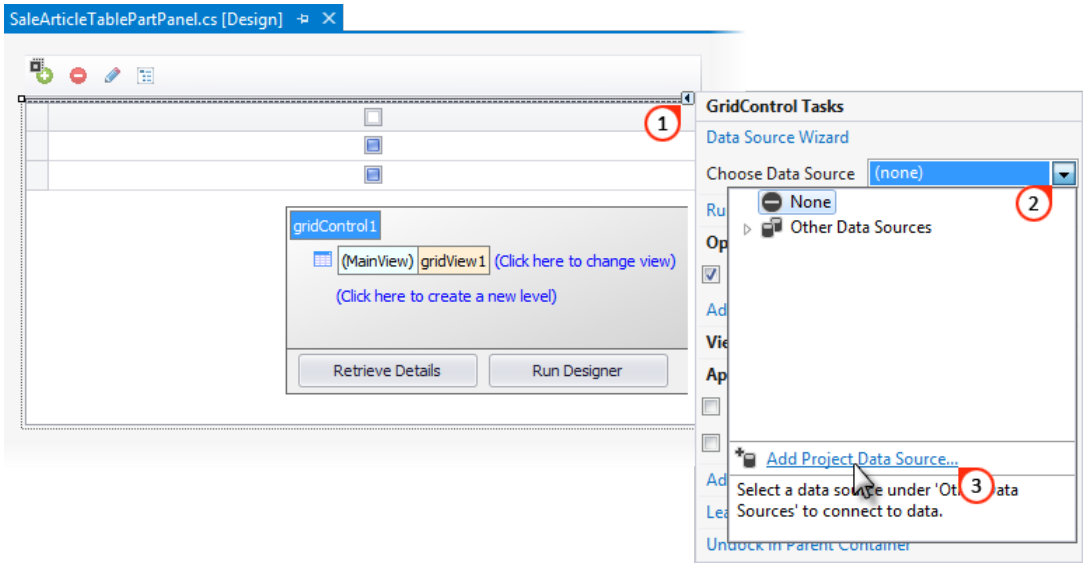
In the module project create new User Control object, derive it from *BaseTablePartGridPanelcontrol element,* (from *Ultima.Client.Controls namespace*), implement *ITablePartEditor<T>* interface:

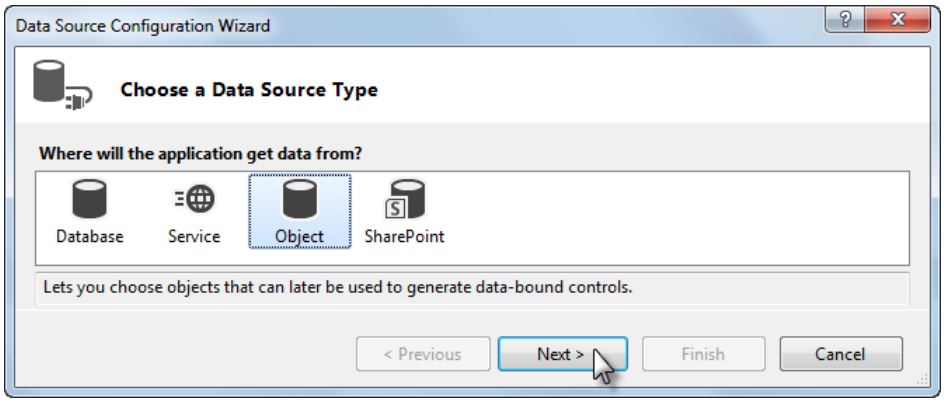
```
public partial class SaleArticleTablePartPanel : BaseTablePartGridPanel,
       ITablePartEditor<ArticleTablePartRow>
{
       public SaleArticleTablePartPanel()
       {
              InitializeComponent();
       }
}
```

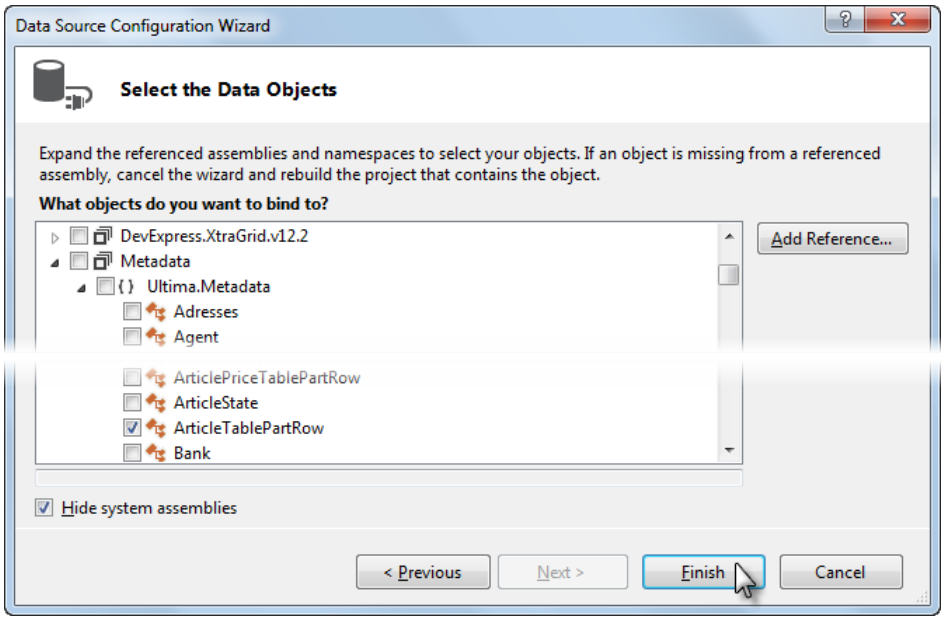


Also set a table part type for *BaseTablePartGridPanel control element* through parameter *TablePartType – ArticleTablePartRow*.

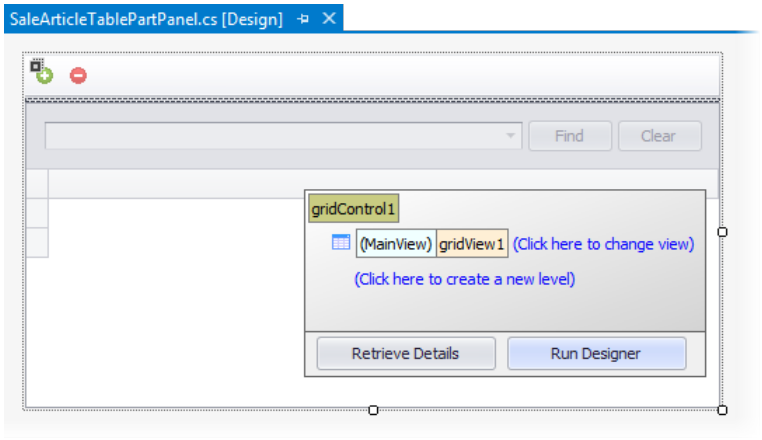
For data mapping of plate part additionally place *GridControl* control element of DevExpress library and connect the same *Ultima.Metadata.ArticleTablePartRow as* metadata object to it as the data source to display table part data :



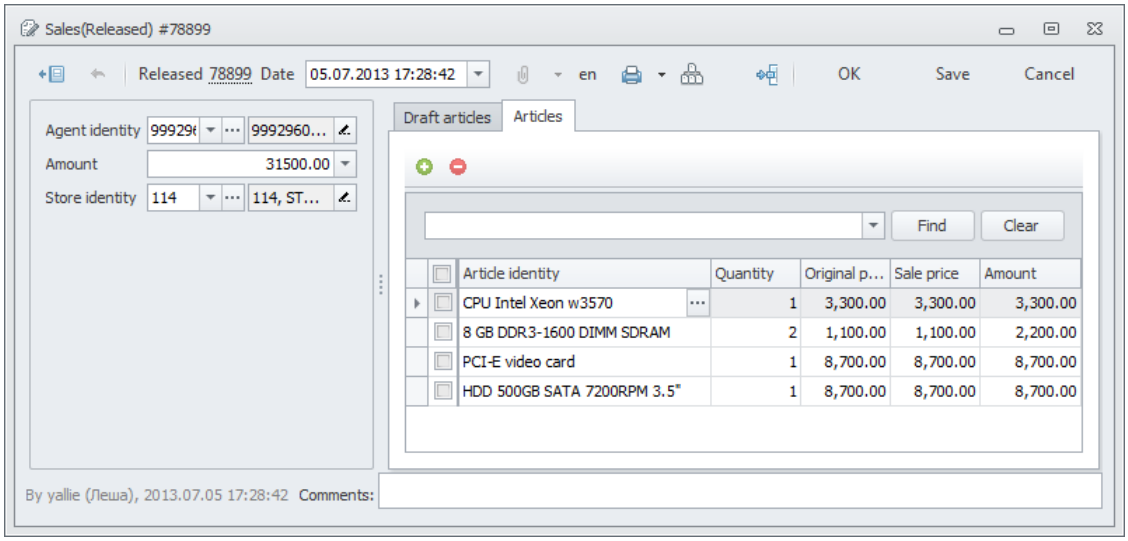




Customize the created control element:



Upon completion compile a project, copy the created libraries into the module folder in the client application *Client/ClientModules/TradeTestSolution*, reload metadata and open the document edit form:



Custom filter

If necessary it is possible to add the element to the filter of a list form by means of `GridPanel.FilterControl.AdditionalFilterControls`.

The event *CustomFilterEventArgs* has a *FilterActive* parameter. For filtration at the switched-on general filter, it is necessary to turn on the filter in the handler of an event at *FilterActive = true*:

```
private void GridPanel_ApplyCustomFilter(object sender,
Client.Controls.CustomFilterEventArgs args)
{
	if (args.FilterActive && OnlyDocumentArticlesChk.Checked)
	{
		var articleList = Articles.OfType<dynamic>().Select(a => (long)
a.ArticleID).ToIDList();
		args.AddFilter<Article>(ar => articleList.Contains(ar.ID));
	}
}
```

To reset the filter there is a *ResetFilters* event:

```
GridPanel.ResetFilters += GridPanel_ResetFilters;

private void GridPanel_ResetFilters(object sender, EventArgs e)
{
	OnlyDocumentArticlesChk.Checked = false;
}
```

Commands

Module command is an element, which can be added to the user interface (main menu) in the user interface setting form. During loading of the module, the kernel polls all classes, derived from *BaseModule* class, to get the list of commands.

Each module can export commands using two methods:

- having marked the method with *Command* attribute;
- having implemented *GetCommands* method, which must return a collection of commands (objects of *Command* class).

We will create one more command as an example for opening of the list form of dictionary Vehicle (in addition to existing one).

For that purpose create a new object Class in the project module, which will be called *TradeTestSolutionModule*, derive it from *BaseModule class* (from *Ultima.Client namespace*) and implement all of its methods:

```
[Export(typeof(IModule)), PartCreationPolicy(CreationPolicy.Shared)]
public class TradeTestSolutionModule : BaseModule
{
	public override string Description
	{
		get { return "Commands of the Trade test solution module"; }
	}

	public override string Name
	{
		get { return "Trade test solution"; }
	}
```

```
        public override System.Resources.ResourceManager ResourceManager
        {
                get { return Resources.ResourceManager; }
        }
}
```

Now when our module is loaded, the kernel will receive all commands described in *TradeTestSolutionModule* class.

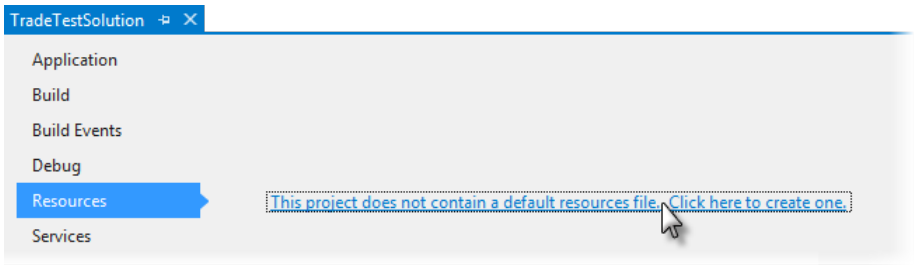
Mark new command using *Command*attribute :

```
[Command("GUID", "Name", "Description", "Category")]
public void CommandName(object sender, CommandEventArgs args)
{
}
```

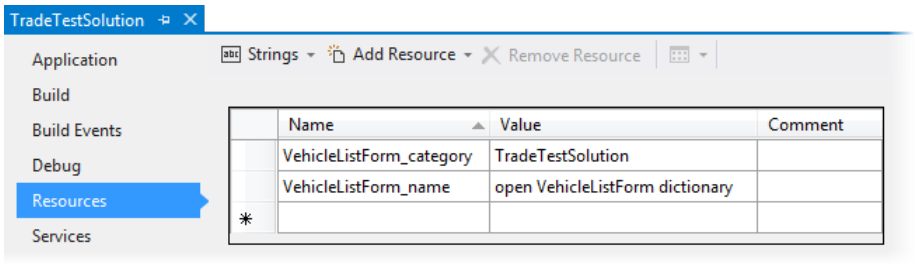
*Command*attribute has the following parameters:

- *GUID* – global ID of the format {6F9619FF-8B86-D011-B42D-00CF4FC964FF};
- *Name* – a name of the resource that keeps the command name;
- *Description* – a name of the resource that keeps the command description;
- *Category* – a name of the resource that keeps the command category. The category can keep description of tree-like structure. The category can store the tree structure description, the parents and children will be separated in this case with symbol "|", and the category itself will have the format "level 1|level 2|level 3".

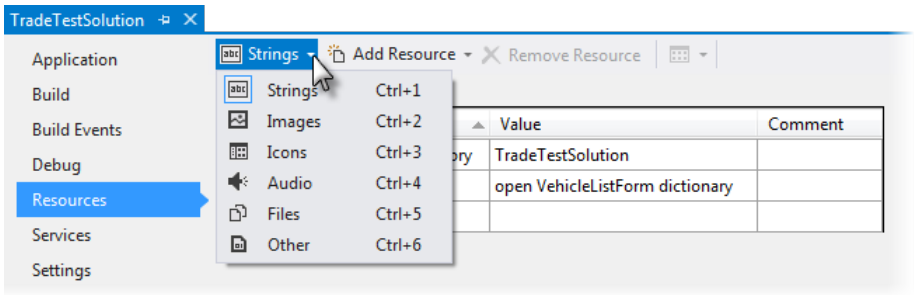
Create corresponding resources for the command in the properties of the module project. During the first access, the file of resources will have to be created:



Names of the resources *Name* will be used in the command:



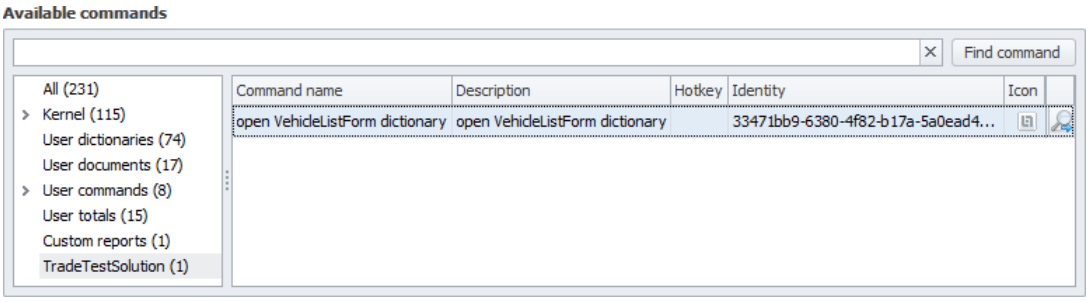
Various (not only text) objects can act as resources:



Now, having created the resources, implement the method, opening previously created list form of the dictionary Vehicle, and mark it with *Command* attribute:

```
// In the example, one and the same resource is used
// for command Name and Description.
[Command("{33471BB9-6380-4F82-B17A-5A0EAD4DDF09}", "VehicleListForm_name",
       "VehicleListForm_name", "VehicleListForm_category")]
public async void OpenVehicleListForm(object sender, CommandEventArgs args)
{
       await DictionaryHelper.BrowseRecords<Vehicle>();
}
```

Compile a project, copy the created libraries into the module folder in the client application *Client/ClientModules/TradeTestSolution*, reload metadata and find the created command in the user interface setting form in the specified category:



Custom screen forms

It is possible to create a screen form for the solution of any task. For that purpose we create new Windows Form and derived it from the class *CommonForm* (from namespace *Ultima.Client*):

```
public partial class CustomForm : CommonForm
{
       public CustomForm()
       {
              InitializeComponent();
       }
}
```

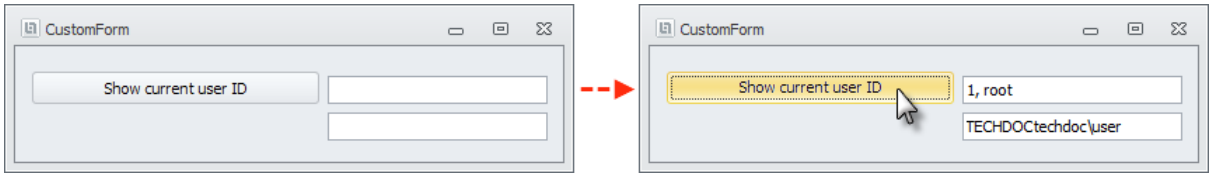
In order to have a possibility to open the created screen form in the client application, add corresponding command into previously created class, derived from *BaseModule* class:

```
[Command("{BB8B0BA1-EE2B-4102-BB1C-BA958CA0F303}", "Name", "Descr", "Category")]
public void OpenMyClientModuleForm(object sender, CommandEventArgs args)
{
       new CustomForm().ShowChild();
}
```

Now using the created form, one can get for instance information about current user:

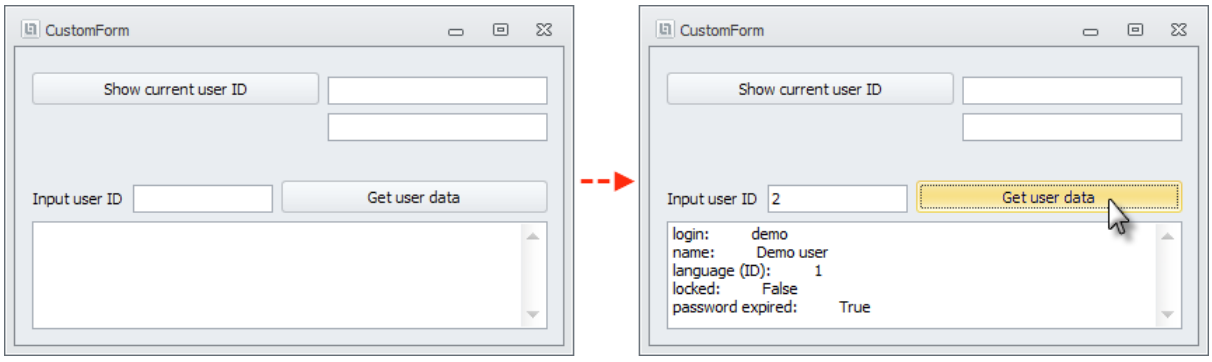
```
[Import]
private IUserManager UserManager { get; set; }

private void simpleButton1_Click(object sender, EventArgs e)
{
       /// We obtain the data on current user.
       textEdit1.Text = UserManager.CurrentUserID + ", "
              + UserManager.CurrentUser.RealUserName;
```

```
        /// We obtain information about user's computer.
        textEdit2.Text = UserManager.CurrentUser.MachineName +
                UserManager.CurrentUser.OsUserName;
}
```



The same information can be obtained by accessing directly the table of users in the database.

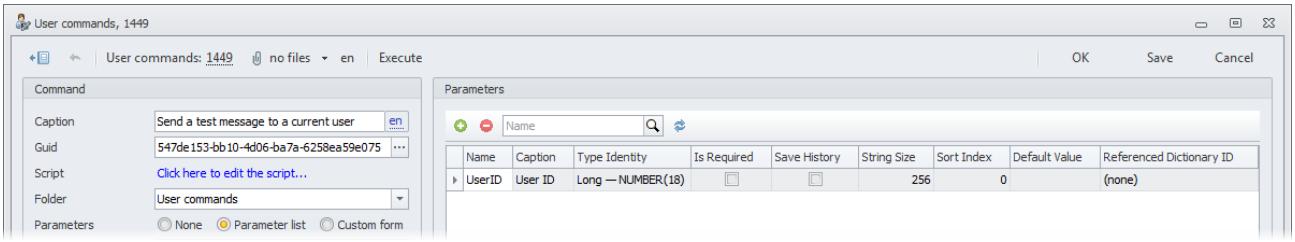
```
[Import]
private ITableSource DataContext { get; set; }

private void simpleButton2_Click(object sender, EventArgs e)
{
        var id = long.Parse(textEdit3.Text);
        var users =
                from u in DataContext.GetTable<User>()
                where u.ID == (id == null ? 1 : id)
                select u;
        var user = users.Single();

        if (user != null)
        {
                memoEdit1.Text = "login:\t" + user.Login + "\r\n" +
                        "name:\t" + user.Name + "\r\n" +
                        "language (ID):\t" + user.LangID + "\r\n" +
                        "locked:\t" + user.IsLocked + "\r\n" +
                        "password expired:\t" + user.IsExpired;
        }
}
```



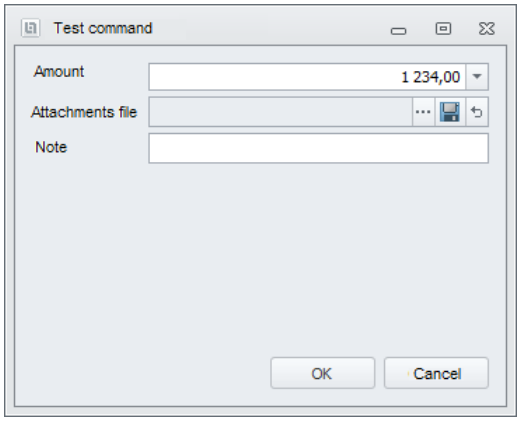
Query forms for the parameters of interactive commands

If necessary it is possible to request the user who is carrying out interactive command to enter values for additional parameters and use them in the future when they run the script. In this case, execution of the command will be preceded with opening of the form for entry of additional parameters.

Additional parameters of interactive command Parameter s*can be retrieved using* standard form (flag *Parameters list*), generated by the system, previously having added them to the list of the same name*Parameters*:



Or they can be requested using independently designed form (flag *Custom form*), in this case there is no need to add parameters to the *Parameters* list, however it is necessary to design a form of parameters independently. The standard (template) form of parameters looks like as follows:



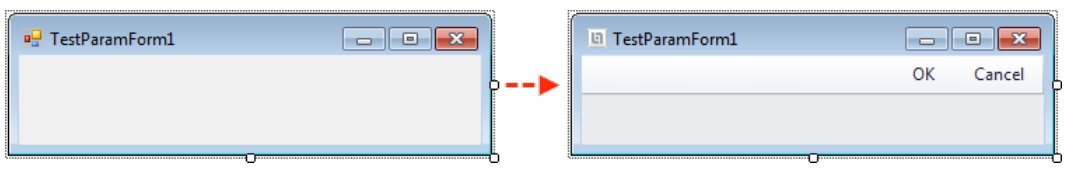
- "OK" button (combination of keys Ctrl + Enter) — closes a form of inquiry of parameters and starts command execution;
- "Cancel" button — closes the form request parameters and cancels the run command.

Possibilities of a sample form of parameters are limited, but this form doesn't demand programming. If the team needs the parameter of non-standard type or if the form of parameters has to display additional data, the form of parameters should be done independently.

We will consider implementation of the second, more difficult option, on the example of the user command which sends the message to the user who executing it. In the message text it is necessary to display the user name corresponding to the identifier entered in the form of additional parameters.

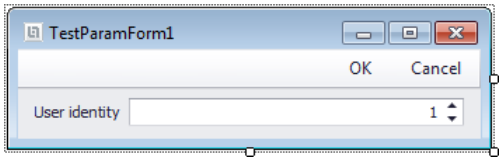
In the project of the module we create a new object of Windows Form, we inherit it from a class *BaseParamForm* (from namespace *Ultima.Client.ParamForms*) also we determine form parameters:

```
[ParamForm("61A32313-4DA6-495F-8062-3CF60ED38EDE","The first test form",
       "Enter parameters in the first test form")]
public partial classTestParamForm1: BaseParamForm
{
       public TestParamForm1()
       {
              InitializeComponent();
       }
}
```



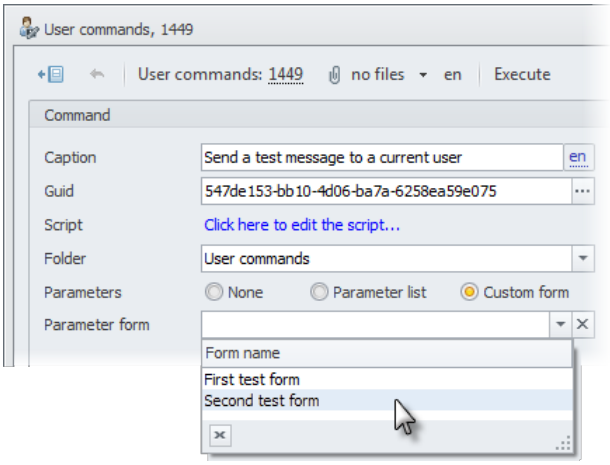
In the property *Parameters* of *BaseParamForm class*under *RecordID key* the object code from which the command was started is transfered.

Add *SpinEdit control element* to the screen form to input of the user identifier:




```
//Receive the value entered in SpinEdit control element.
Protectedoverride void GetParameters()
{
	Parameters[”UserID”] = spinEdit.Value;
}
```

And creation of parameter entry form is completed. Compile a project, copy the created libraries into the module folder in the client application *Client/ClientModules/TradeTestSolution*, reload metadata and transfer to command edit. Having set *Custom form flag* for property *Parameters* select the created additional parameter entry form from *Parameter form list*:

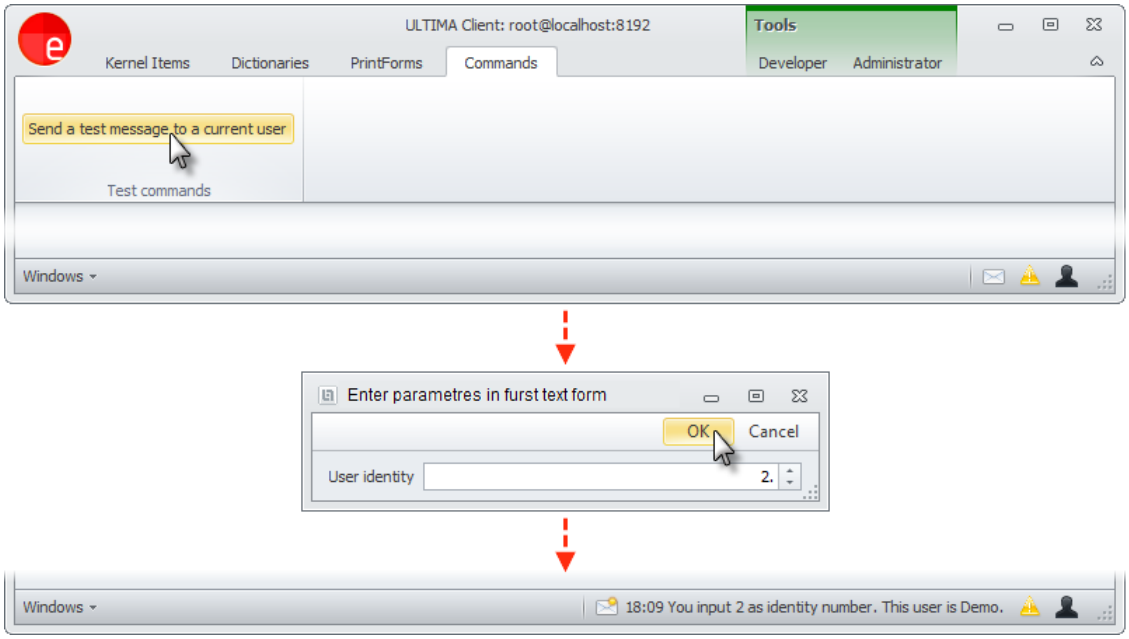


After that it is possible to pass to script edit which will send a message to the user:

Script text | Resources | Generated Text (read-only) | Generated Resx (read-only) | MEF Cache (read-only)

```
using System.Collections.Generic;
using System.ComponentModel.Composition;
using Ultima.Client;
using Ultima.Client.Actions;
using Ultima.Server.Data;
using Ultima.Collections;

namespace Ultima.Scripting
{
    public partial class SendTestMessageToCurrentUser
    {
        [Import]
        private IUserManager UserManager { get; set; }

        [Import]
        private IUserMessages UserMessages { get; set; }

        public void Execute(IDictionary<string, object> parameters, IList<ClientAction> clientActions)
        {
            decimal dUserId = default(decimal);

            // try to get user ID from parameters
            if (!parameters.TryGetValue("UserID", out dUserId))
            {
                return;
            }

            long userId = System.Convert.ToInt64(dUserId);

            var userName = "Unknown user";

            // get current user name using UltimaDbManager
            using (var db = new UltimaDbManager())
            {
                var query = "select Name from KERNEL.USERS where ID = :vID";
                userName = db.SetCommand(query, db.Parameter("vID", userId)).ExecuteScalar<string>();
            }

            // send message for current user
            UserMessages.CreateUserMessage("You input {0} as identity number. This user is {1}.", userId, userName);
        }
    }
}
```

Obtain user ID from additional parameter form, request the user name corresponding to this ID and send a message:



The data entered into the parameter form can be checked for a validity by means of the following option *GetParameters method*:

```
GetParameters(string title,
              IEnumerable<ScriptUserParameter> parameters,
              IDictionary<string, object> paramValues,
              Action<ParameterValidationHelper> validationMethod)
```

ParameterValidationHelper class allows transfering parameter values to the validation method, and back – results of check with any necessary level of detail, for example, to highlight the wrong parameters in the form through *ErrorProvider*, for example:

```
ParameterValidationHelper =>
{
       var x = ParameterValidationHelper.Parameters["Password"];
       if (x.ToString().Length < 5)
       {
              ParameterValidationHelper.ReportError(”Password”, “, “Password is too
short");
       }
}
```

Application main form

The window, which contains the main menu and tabs of all other commands and windows, is called as the main form of the client application. Life cycle of the application is tied to this window: when the user closes the window, the application comes to the end. By default, the main window of the application has the interface described in the user manual.

If the application needs the main form which looks in a different way, it is possible to realize it independently. We recommend to use this only in exceptional cases, as for this purpose it is necessary to refuse templates of the main menu, standard modules (basic module and the developer module) and other useful functionality, provided by the standard main form.

For independent realization of the main application form the following is required:

- To develop the form of the desired type using the Visual Studio forms designer
- To realize interface methods IMainForm in the form class.
- To export the form: [Export(typeof(IMainForm)), PartCreationPolicy(CreationPolicy.Shared)]
- Additionally, it is necessary to create a new class (commonly called as Program), which implements the entry point to the application. It is necessary to import our form in it:

```
[Import]
private Lazy<IMainForm> MainForm { get; set; }
```

- We create the method in this class, exporting the entry point (it has to be exactly one):

```
[Export(ContractNames.EntryPointMethod)]
public Ultima.Client.CloseReason Main(string[] commandLineArgs)
{
   Application.Run(MainForm.Value as Form);
   return MainForm.Value.CloseReason;
}
```

It is important that the method exporting the entry point had a certain signature. It has to have the only one parameter — the massif of lines, and the returned value has to be the type of `Ultima.Client.CloseReason`.

The behavior when closing the form is defined by CloseReason value:

- Normal — the application is closed
- Restart — the application is restarted.

Mobile application

The client mobile application is developed in C# using Xamarin http://xamarin.com/ with employment of mobile interfaces, which are compiled into a separate library mobilemetadata.dll (ultimalib.dll and mobileinterfaces.dll libraries are required too).

There are two connection options for Android client: Zyan and web-services.

Two strings are generated for web-services: ordinary WebServices.dll and transferable WebServices.Portable.dll. For Android-application, the second assembly should be selected and PCL-client should be used for connection to the database. An example of PCL-client, which works on PC:

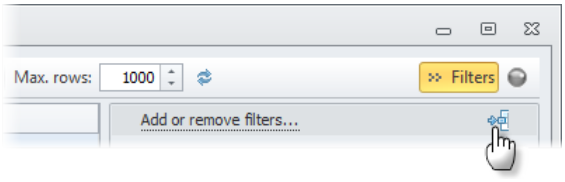
```
// Compile using:
// c:\Windows\Microsoft.NET\Framework\v4.0.30319\csc test.cs /r:WebServices.dll
// /r:ServiceStack.Client.dll /r:ServiceStack.Interfaces.dll /r:System.Runtime.dll
// /r:ServiceStack.Pcl.Net45.dll

using System;
using ServiceStack;
using Ultima;
using Ultima.WebServices;

class Program
{
	static void Main()
	{
		Net40PclExportClient.Configure();
		var client = new JsonServiceClient("http://localhost:8337/");

		var response = client.Get<GetNowResponse>(new GetNow());
		Console.WriteLine("response: {0}, null: {1}", response, response == null);

		if (response != null)
		{
			Console.WriteLine("nowResponse: {0}, isoTime: {1}", response,
response.IsoTime);
		}
	}
}
```

For mobile application, the second call Configure will be in the first string. Besides, instead of Get<Response>(), await GetAsync<Response>() should be used.

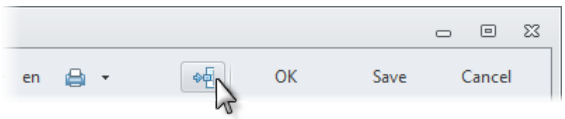
System tools for setting of the appearance of screen forms_2

The layout of standard screen forms of created dictionaries and documents can be customized.

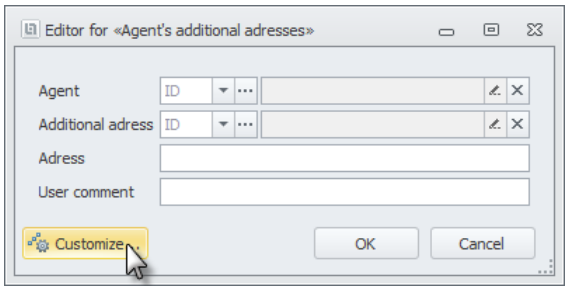
For example, the layout of the filter of records can be changed for the dictionary (log of documents) list form.



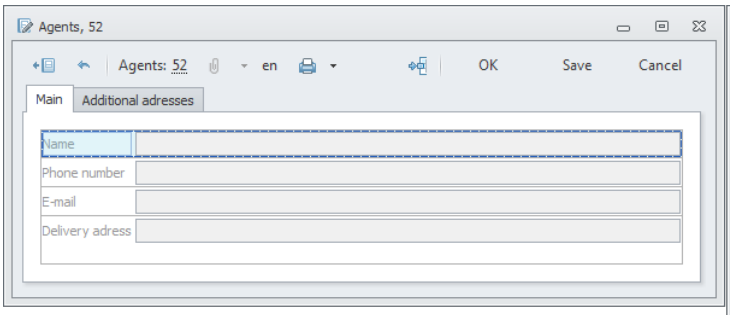
The layout of the edit form for dictionary records (document) can be changed.

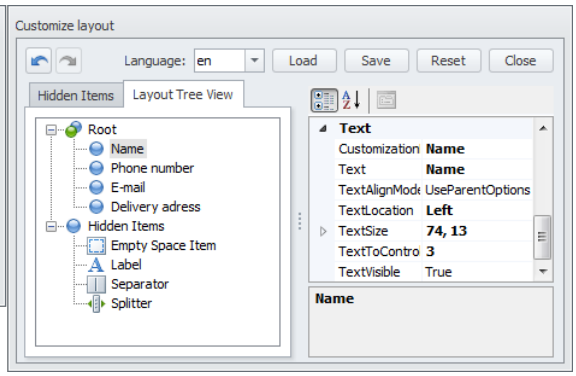


Or layout of the edit form for the link table records.



All changes are made using single tool – forms for modification of the layouts "Customize layout":



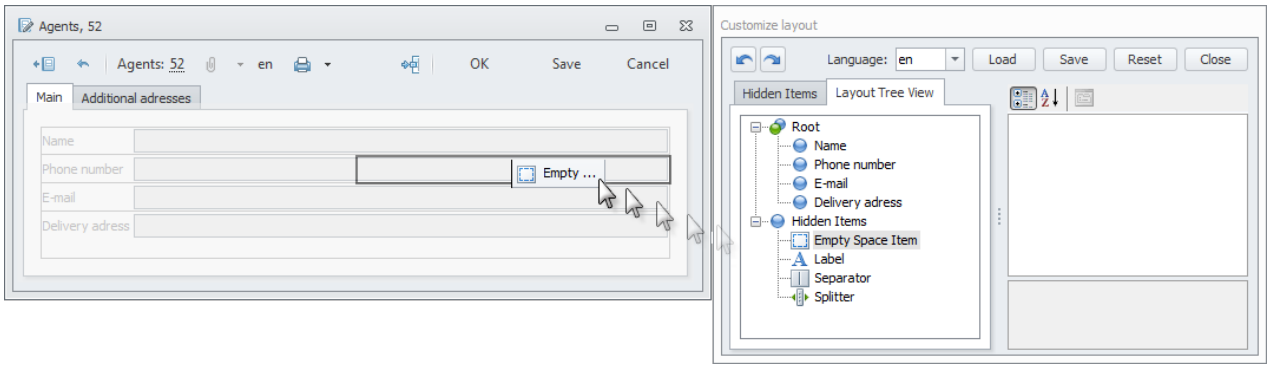


- the changes being made can be revoked or repeated, in case of revocation, using corresponding buttons ;
- there is own layout for each of system languages. Selection of the language is carried out in the control element "Language";
- "Load" button performs loading of the layout for the language selected in the field *Language*. If no layout has been created yet for loaded language, but the form has the layout for another language - this existing layout will be loaded, however the names of the metadata objects properties will be loaded corresponding to the layout language;
- "Save" button performs saving of the edited layout for the language selected in the field "Language". It allows, e.g. constructing the layout only one time for one of the system languages and then just localizing it for another language;

If the layout of the screen form is loaded for the system language, e.g *English* (*en*), the changes are made to it, and then another system language is selected, e.g. *Russian* (*ru*) and the layout is saved, the screen form layout for *English* language will remain unchanged, and the layout for *Russian* language will be changed.

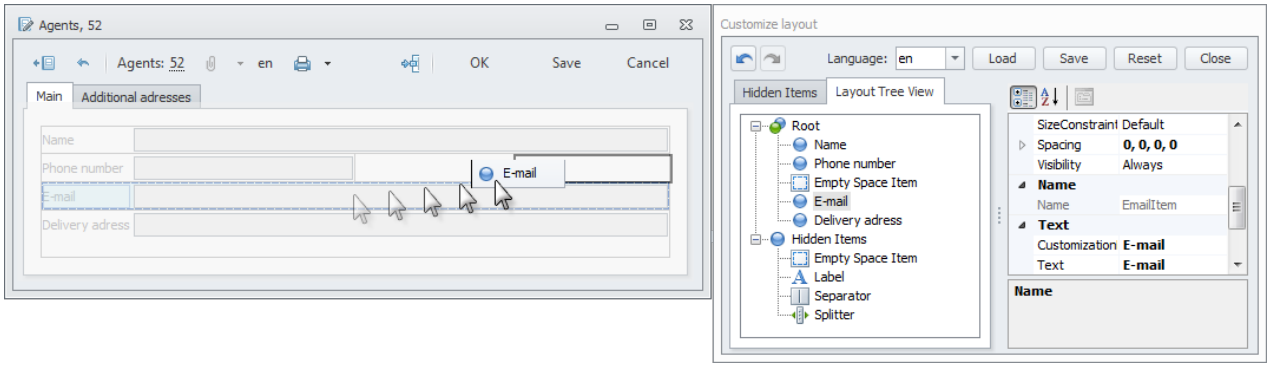
- "Reset" button resets all changes ever made to the layout of the language selected in the field "Language";
- "Close" button closes the form for layout modification, not saving the changes made;
- in the tab "Layout Tree View", the elements are grouped into tree view, which are used (or available for use) in the screen form layout:
  - the elements already located on the screen form are in *Root* branch;

- in *Hidden Items* branch, the elements are present, which can be used in the screen form, just by drag-and-drop on its layout. It can be both the properties removed from the layout of dictionary property (document head) and standard elements:

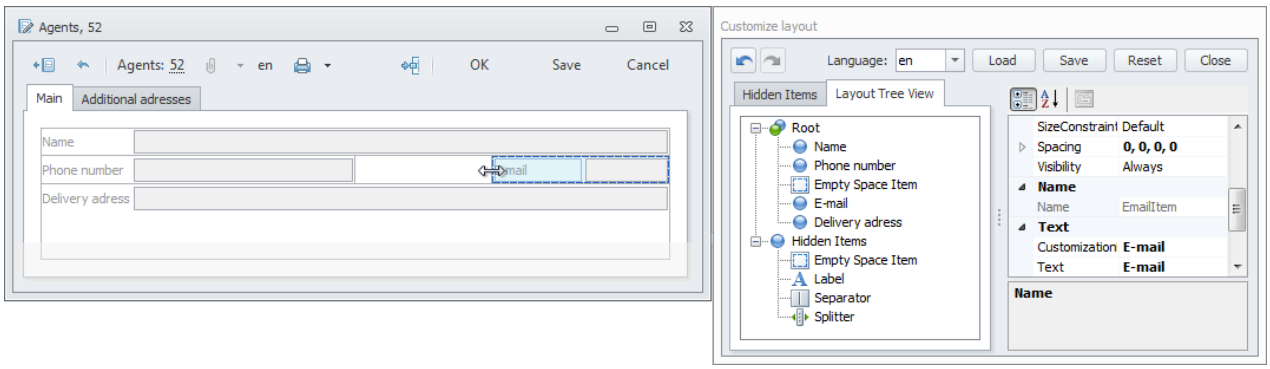


- the list of properties of the element, selected on the screen form layout or in *Root* branch in the tab "Layout Tree View" of form "Customize layout", is located on the right in the tab "Layout Tree View". The properties can be arranged by groups or in alphabetic order ;
- the content of tab "Hidden Items" duplicates the content of *Hidden Items* branch in the tree in the tab "Layout Tree View".

The elements on the layout of screen form can be dragged and dropped:



And their size can be changed:



If standard screen forms and functionality of layout editor are insufficient, own list form or edit form can be constructed independently in any .NET compatible language in any development environment suitable for that.

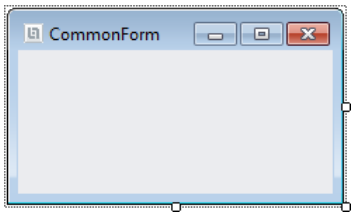
The screen form should be created in the shared module of client application.

Ultima control elements

The system Ultimate AEGIS® provides to the applied developer a number of *classes*, *forms* and *elements of management* for realization of screen logic and the user interface.

In order to get access to Ultima control elements in Visual Studio , we have to create a folder (e.g. Ultima) in the Toolbox and add the control elements into it from the library BaseClientLibrary.dll (Choose Items -> Browse (at the tab .NET Framework Components) -> BaseClientLibrary.dll).

CommonForm



CommonForm (from *Ultima.Client* namespace) is a common parent for all screen forms of a client application. The form supports saving coordinates and sizes, saving and loading form's settings, displaying the form's status.

Also, the form's property *AutoSetTabOrder* allows setup a switching between controls of the form using the Tab key:

- if *true* (default value), the switching order is specified automatically (top to down , right to left);
- if *false*, the order specified by application programmer is applied.

CommonForm basic functions also include the *ReadOnly* property. The read-only mode is active during background processes: *DisplayFormStatus(Strings.Loading)* or *BaseEditForm.SaveAndLoadRecord*.

Checking *ReadOnly* = *true* leads to a recursive switching between the form's elements. Unchecking *ReadOnly* = *false* return the settings to the original state (in the process, if a control was initially in a *ReadOnly* mode, it will not be suddenly unblocked).

Since in WinForms, there is no a regular *ReadOnly* property similar to the *Enabled* flag, the mode is supported with the help of a set of adapter-classes. The base set supports the most spread controls: TextEdit, MemoEdit, UltimaTextEdit, CheckEdit, RadioGroup, GridControl, various container elements.

To add a support of a new control element, it takes only to define in the client application module an adapter-class that supports the given element:

```
using System;
using System.Windows.Forms;
using DevExpress.XtraEditors;

namespace Ultima.Client.Controls.ReadOnly
{
	[PartCreationPolicy(CreationPolicy.NonShared)]
	internal class ReadOnlyAdapterCheckEdit : ReadOnlyAdapterBase
	{
		public override bool Supports(Control control)
		{
			return control is CheckEdit;
		}

		public override bool IsReadOnly(Control control)
		{
			return (control as CheckEdit).Properties.ReadOnly;
		}

		protected override void SetReadOnly(Control control, bool readOnly)
		{
			(control as CheckEdit).Properties.ReadOnly = readOnly;
		}
	}
}
```

The adapter will be automatically recognized by MEF catalogs and used if needed. To do this, it is sufficient to inherit the class from *ReadOnlyAdapterBase* or to make the class to explicitly export the *IReadOnlyAdapter* interface implementation. In addition, the NonShared creation policy must be specified for the class.

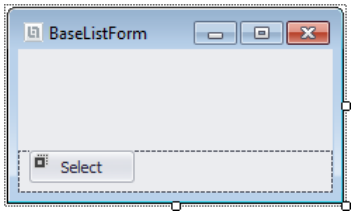
When inheriting from the *CommonForm*, the following methods and properties of the form's class may be helpful to application programmer:

- *ShowChildAsync()* – an asynchronous variant of the *ShowChild* method. Displays the form as MDI child of the main form and waits till its settings are loaded;
- *AfterLoadSettings* – event that occurs after the form's settings are loaded;
- *AfterSaveSettings* – event that occurs after the form's settings are saved;
- *ResetSettings* – event that occurs after the form's settings are reset.

One also can redefine the following methods and properties of the *CommonForm*:

- *ShowChild()* – show the form as MDI child of the main form;
- *OnLoadAsync(EventArgs args)* – an asynchronous variant of the OnLoad method. If in the applied code, there is a need to make particular server calls while loading a form, e. g., to load a record or read a constant, these calls are to be done within this method;
- *FormSettingsKey* of *string* type – returns a key, under which the form settings are saved. As a rule, this is a class name, but it may contain some extra characters;
- *DisplayFormStatus(string text)* – displays a panel with the message text specified (e. g., "Loading..."):
  - *text* – message text;
- *ClearSettingsBeforeSaving* of *bool* type – returns *true*, if the collection needs to be cleared to save the form's settings;
- *RestoreWindowBounds* of *bool* type – returns *true*, if the form is to restore the window sizes;
- *SaveSettings()* – saves form's settings in the *FormSettings* collection;
- *LoadSettings()* – loads form's settings from the *FormSettings* collection.

BaseListForm



BaseEditForm form (from *Ultima.Client namespace*) is derived from *CommonForm* and is common parent for all edit forms of dictionaries and documents:

CommonForm

BaseListForm

The form is used to display the record list and their choice.

In *BaseListForm form class IRecordBrowser* and *IRecordSelector interfaces are realized*.

In derivation from *BaseListForm form* the following methods and properties of its class can be useful to the application developer:

- *Mode* of *ListFormShowMode type*– returns the mode of the list form which can have one of the following values:
  - *Browse* – record browse mode;
  - *SelectSingle* – single record select mode;
  - *SelectMultiple* – several record select mode;
- *SelectRecord(long? id = null, LambdaExpression filter = null)* – selects a record and returns its identifier or *null*, if record wasn't selected:
  - *id* – a record identifier on which the cursor in the list form will be set (option parameter);
  - *filter* – expression describing the filter which will be used to the records removed in the list form (option parameter);

- *SelectRecords(LambdaExpression filter = null)* – selects several record and returns their identifiers or *null*, if records weren't selected:
  - *filter* – expression describing the filter which will be used to the records removed in the list form (option parameter);
- *Browse()* – opens the list form, and loads records.

It is also possible to redefine the following methods and properties of *BaseListForm form class*:

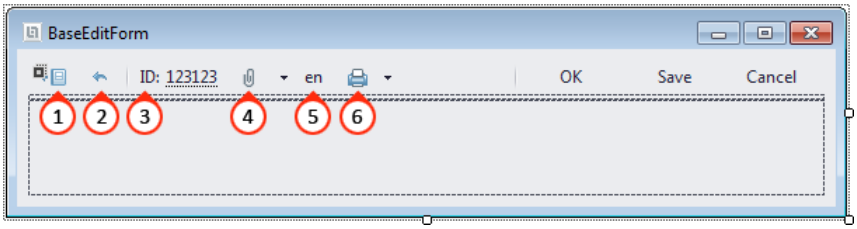
- *SelectedID* of *long type*– returns a code of the selected record if the form was caused to select;
- *SelectedList of* type *IDList* – returns codes of the selected records if the form was caused to select;
- *LoadRecords()* – loads records;;
- *ApplySelectionFilter(LambdaExpression implicitFilter)* – applies an implicit filter expression to the form records list:
  - *implicitFilter* – expression describing the filter used to the form records list;
- *LocateRecord(long id)* – sets the cursor on the specified record, *true* in case of success, differently – *false* returns:
  - *id* – the record identifier.

BaseEditForm

BaseEditForm (from *Ultima.Client* namespace) is inherited from *CommonForm* and is a common parent for all edit forms of dictionaries and documents:

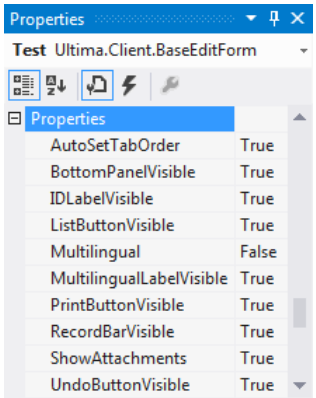
CommonForm
BaseEditForm

The form supports multilinguality, attachments, printing, opening the list form of edited object, cancel and saving changes made. The form includes also a toolbar with all the functions specified:



The control element has the following specific properties:

- *IDLabelVisible* – if *true*, ID of the object opened in the form is displayed (3);
- *ListButtonVisible* – if *true*, the button to open a list form of the object opened in the *BaseEditForm* is displayed (1);
- *MultilingualLabelVisible* – if *true*, multilanguage label is displayed (5);
- *PrintButtonVisible* – if *true*, the print button is displayed (6);
- *RecordBarVisible* – if *true*, the toolbar is displayed;
- *ShowAttachments* – if *true*, the attachments button is displayed (4);



- *UndoButtonVisible* – if *true*, the button to cancel any changes made (but not saved) to the object opened in the form is displayed (2).

In the *BaseEditForm* form class, the *IRecordEditor* interface is implemented, as well as the *DictionaryManager* is imported.

When inheriting from the *BaseEditForm*, the following methods and properties of the form's class may be helpful to application programmer:

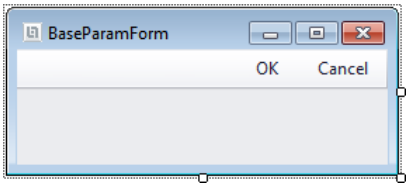
- *DataRecordChanged* – event that occurs after the data of the record being edited have been changed;

- *Print* – event that occurs after the Print button is clicked;
- *Export* – event that occurs after the Export button is clicked;
- *LoadedRecord* – event that occurs after a record is loaded.
- *SaveRecordAndClose()* – saves changes made and closes the form;
- *ID* of *long* type – code of a current dictionary (document) record being edited;
- *DataRecord* of *IBusinessObject* type – a current dictionary (document) record being edited.

One also can redefine the following methods and properties of the *BaseEditForm* class:

- GetAvailableToolbarLinks() – allows adding extra buttons to the toolbar;
- *ValidateRecord(ValidationErrorCollection errors)* – allows redefining the record check before saving in the database:
  - *errors* – collection of errors;
- *ShowValidationErrors()* – determines the logic for displaying validation errors in the form, e. g., in order to single out validation errors via a non-routine method;
- *EditRecord(long id)* – opens the specified record to edit in the modal mode:
  - *id* – record ID;
- *InsertRecord(IDictionary<string, object> parameters)* – initiates creation of a record in the modal mode:
  - *parameters* – parameters of the newly-created record;
- *CloneRecord(long baseId)* – clones the specified record in the modal mode:
  - *baseId* – ID of the record to be cloned;
- *ViewRecord(long id)* – displays the specified record as "read-only" in the modal mode:
  - *id* – record ID;
- *BeginEditRecord(long id)* – opens the specified record to edit in the modeless mode:
  - *id* – record ID;
- *BeginInsertRecord(IDictionary<string, object> parameters)* – initiates creation of a record in the modeless mode:
  - *parameters* – parameters of the newly-created record;
- *BeginCloneRecord(long baseId)* – clones the specified record in the modeless mode:
  - *baseId* – ID of the record to be cloned;
- *BeginViewRecord(long id)* – displays the specified record as "read-only" in the modeless mode:
  - *id* – record ID;
- *Modified* of *bool* type – redefines the check, if the record was modified;
- *InternalCreateRecord(IDictionary<string, object> parameters)* – redefines creation of a new dictionary (document) record:
  - *parameters* – parameters of the newly-created record;
- *InternalLoadRecord(long id)* – redefines loading of the new dictionary (document) record:
  - *id* – record ID;
- *InternalSaveRecord()* – redefines saving of the new dictionary (document) record;
- *OnDataRecordPropertyChanged(string propertyName)* – the method is called when the specified property of a dictionary (document) record has been changed:
  - *propertyName* – name of property;
- *EndEdit()* – applies changes introduced by user to the current record loaded. E. g., if user is editing the form's text field, the field's value is copied to the current record;
- *RejectChanges()* – cancels changes introduced by user to the record and returns the record to its original state;
- *BrowseRecords()* – shows a list form of a record opened in the edit form.

BaseParamForm



The form *BaseParamForm* (from namespace *Ultima.Client.ParamForms*) is derived from *CommonForm* is common parent for all edit forms of parameters query of interactive commands:

CommonForm
- *BaseParamForm*

The form is used to create its own forms of parameters query of interactive commands (the process is described in detail in chapter Form of additional command options).

When you deriving from the form *BaseParamForm* the following methods and properties of its class can be useful to the application developer:

- *RequestParameters(string title, IDictionary<string, object> parameters)* – allows filling a collection of parameters with values. It turns *true*, if the “OK” button was pressed, and *false* otherwise:
  - *title* – heading of the parameters form;
  - *parameters* – collection of parameters for filling.

It is also possible to redefine the following methods and properties of a class of the *BaseParamForm* form:

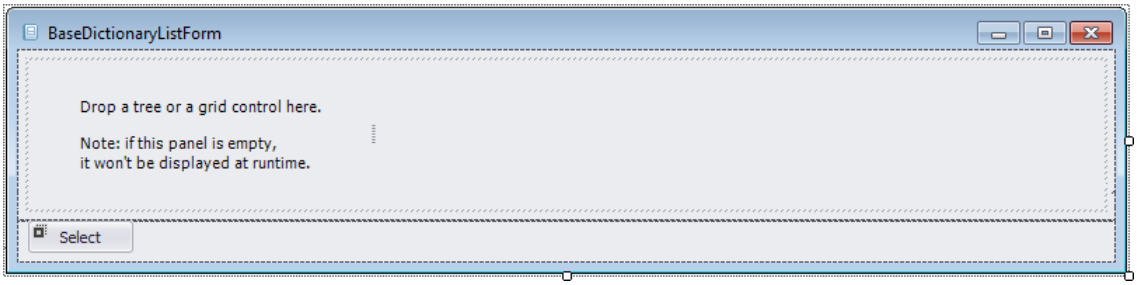
- *GetParameters()* – allows filling the collection of parameters with values;
- *CheckData()* – defines the check of parameters before copying them in the collection of values;
- *OkButtonClick()* – is called when pressing "OK" button;
- *CancelButtonClick()* – is called when pressing "Cancel" button;
- *GetAvailableToolbarLinks()* – allows adding additional buttons in form toolbars (initially the form contains only the “OK” and “Cancel”buttons).

BaseDictionaryListForm

The form *BaseDictionaryEditForm* (from namespace *Ultima.Client.Dictionaries*) is derived from *BaseEditForm* and is common parent for all edit forms of dictionaries:

CommonForm
- *BaseListForm*
  - *BaseDictionaryListForm*

On a form the element of management *SplitContainer*, is placed, which keeps the position in settings of the form. The left panel of the container is reserved under placement of the filter (the table or a tree) if to leave this area empty – it will be hidden in the total form.



The form is used for creation of list forms of the dictionary with screen logic, significantly other than the list form by default. To create a master-detail of the interface on the basis of a list form it is recommended to use *BaseFlatDictionaryListForm* or *BaseTreeDictionaryListForm* by default.

In the class of *BaseDictionaryListForm* form, *DictionaryManager* is imported.

The class of *BaseDictionaryListForm* form has the following properties:

- *CustomFormText* of type *bool* – returns *true*, if the property *Text* of the form has to remain so as it has been specified at its development. Otherwise the system will create the form heading independently.

TIt is also possible to redefine the following methods and properties of a class of the *BaseDictionaryListForm* form:

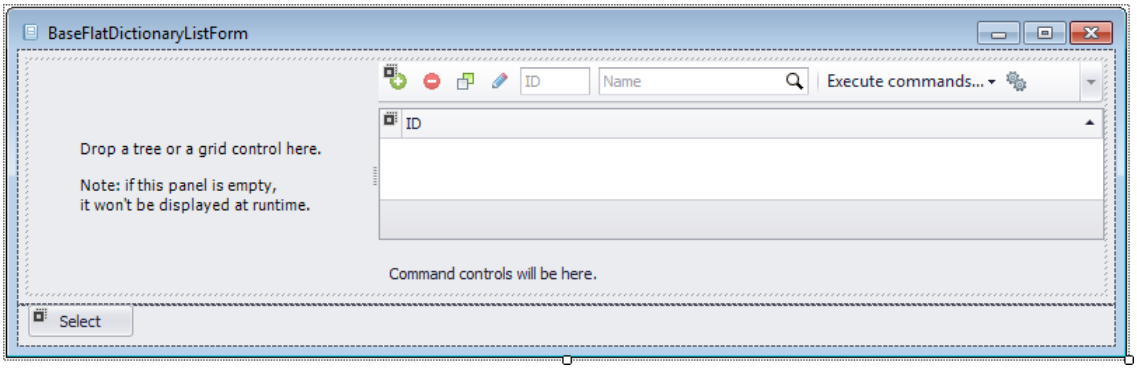
- *DictionaryType* of the type *Type* – type of the dictionary which is displayed by the form.

BaseFlatDictionaryListForm

BaseFlatDictionaryListForm form (from *Ultima.Client.Dictionaries namespace*) is derived from *BaseDictionaryListForm* and is common parent for all list forms of flat dictionaries:

CommonForm
- *BaseListForm*
  - *BaseDictionaryListForm*
    - *BaseFlatDictionaryListForm*

The most frequent reason for creation of own list form of the dictionary is implementation of the master-detail interface. Therefore, *BaseFlatDictionaryListForm* hosts already *SplitContainer control element derived fromm BaseDictionaryListForm* , on which right panel *DictionaryGridViewPanel*, control element is located, designed to view the list of flat dictionary records. The element contains a toolbar with the entire standard functionality - set of columns, filters, etc. Only left panel, which is reserved for location of the filter, is available for use to the application developer. If this area is left empty – it will be hidden in the final form:



To implement own list form of flat dictionary, it should be derived from *BaseFlatDictionaryListForm* form and *IRecordBrowser<T>* and *IRecordSelector<T>* interfaces should be implemented, where *T* is a dictionary type. The system will search for the form implementing *IRecordBrowser<T>* interface to display the list of records, and for selection of the records it will search for the form implementing *IRecordSelector<T>*. If no such form appears to be in the system, the basic dictionary list form will open. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator. The process of creation of own dictionary list form is detailed in the chapter List forms of dictionaries.

The class of *BaseFlatDictionaryListForm* form implements the following methods and has the following properties:

- *DictionaryType*, type *type* returns a dictionary type;
- *SelectedID*, type *long* returns ID of dictionary record selected in *DictionaryGridViewPanel* control element;
- *SelectedList*, type *IDList* returns a list of IDs of dictionary records selected in *DictionaryGridViewPanel* control element;
- *LoadRecords()* loads the dictionary record into *DictionaryGridViewPanel* control element;

- *GridPanel*, type *DictionaryGridViewPanel* it returns *DictionaryGridViewPanel* control element. Using this property, the values can be set for instance for *DictionaryGridViewPanel*, properties, while customizing the interface of this control element:

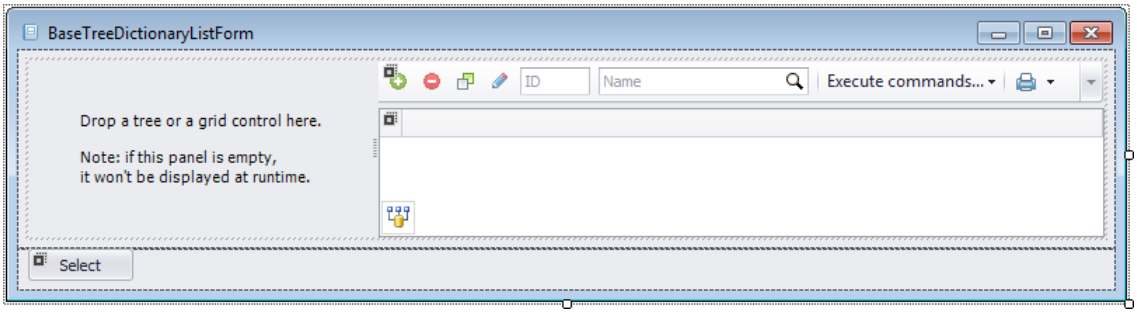
```
GridPanel.Properties.LimitCounterVisible = false;
GridPanel.Properties.GroupButtonVisible = false;
GridPanel.Properties.PrintButtonVisible = false;
GridPanel.Properties.IDEditVisible = false;
GridPanel.Properties.CommandsMenuVisible = false;
```

BaseTreeDictionaryListForm

BaseTreeDictionaryListForm form (from *Ultima.Client.Dictionaries namespace*) is derived from *BaseDictionaryListForm* and is common parent for all list forms of tree dictionaries:

CommonForm
- *BaseListForm*
  - *BaseDictionaryListForm*
    - *BaseTreeDictionaryListForm*

The most frequent reason to create own list form of the dictionary is implementation of the master-detail interface, which is also called a filter. Therefore, *BaseTreeDictionaryListForm* form already hosts *SplitContainerBaseDictionaryListForm* control element derived from , on which right panel *DictionaryTreeViewPanel* control element is located, designed to view the list of tree dictionary records. The element contains a toolbar with the entire standard functionality - set of columns, filters, etc. Only left panel, which is reserved for location of the filter, is available for use to the application developer. If this area is left empty – it will be hidden in the final form:



To implement own list form of tree dictionary, it should be derived from *BaseTreeDictionaryListForm* form and *IRecordBrowser<T>* and *IRecordSelector<T>* interfaces should be implemented, where *T* is a dictionary type. The system will search for the form implementing *IRecordBrowser<T>* interface to display the list of records, and for selection of the records it will search for the form implementing *IRecordSelector<T>*. If no such form appears to be in the system, the basic dictionary list form will open. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator.

The class of *BaseTreeDictionaryListForm* form implements the following methods and has the following properties:

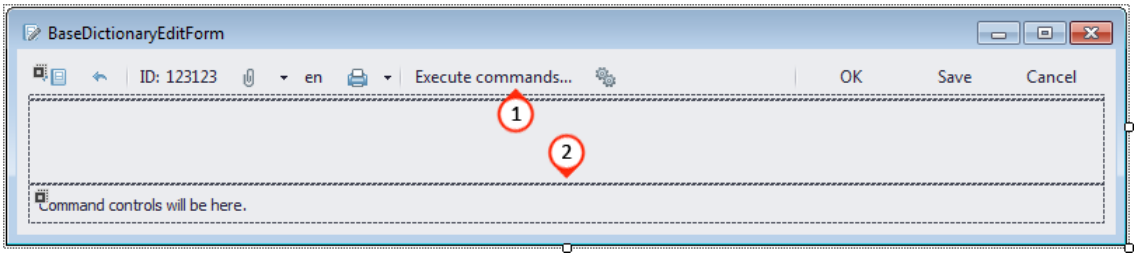
- *DictionaryType*, type *type* returns a dictionary type;
- *SelectedID*, type *long* returns ID of dictionary record selected in *DictionaryTreeViewPanel* control element;
- *SelectedList*, type *IDList* returns a list of IDs of dictionary records selected in *DictionaryTreeViewPanel* control element;
- *LoadRecords()* loads the dictionary record into *DictionaryTreeViewPanel* control element;
- *TreePanel*, type *DictionaryTreeViewPanel* returns *DictionaryTreeViewPanel* control element. Using this property, the values can be set for instance for *DictionaryTreeViewPanel* properties, while customizing the interface of this control element.

BaseDictionaryEditForm

The form *BaseDictionaryEditForm* (from namespace *Ultima.Client.Dictionaries*) is derived from *BaseEditForm* and is common parent for all edit forms of dictionaries:

CommonForm
- *BaseEditForm*
  - *BaseDictionaryEditForm*

The form has implemented support for the commands on dictionary record. The form contains also a toolbar with the functionality enumerated and derived from *BaseEditForm*:



To implement own edit form for dictionary records, it should be derived from *BaseDictionaryEditForm* form, and *IRecordEditor<T>* interface should be implemented, where *T* is a type of dictionary. The system for editing of the dictionary record will search for the form implementing *IRecordEditor<T>* interface. If no such form appears to be in the system, the basic dictionary record edit form will open. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator. The process of creation of own dictionary edit form is detailed in the chapter Edit forms of dictionary records.

The control element has the following specific properties (the properties derived from *BaseEditForm* are described in the corresponding section):
- *CommandsMenuVisible* – if *true*, a button is displayed for the list of commands on dictionary record (**1**);
- *QuickCommandsVisible* – if *true*, a toolbar is displayed for quick access to the commands at the form bottom (**2**).

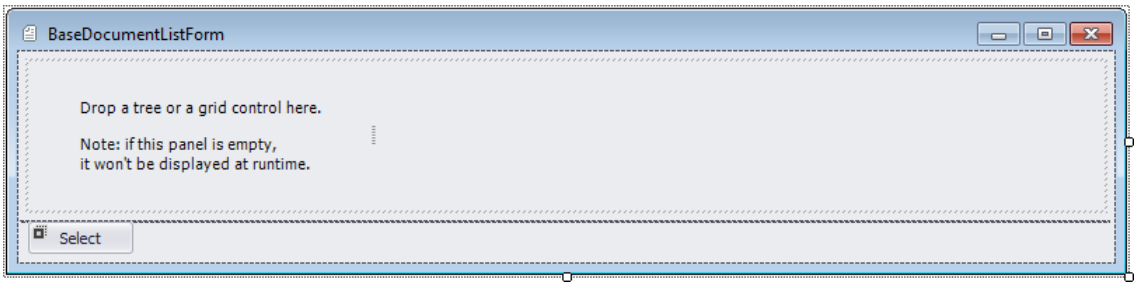
The class of *BaseDictionaryEditForm* form has the following properties:
- *DictionaryType*, type *type* returns a dictionary type;
- *DataRecord*, type *IDictionaryRecord* returns a dictionary record opened in the form.

BaseDocumentListForm

The form *BaseDocumentListForm* (from namespace *Ultima.Client.Documents*) is derived from *BaseListForm* and is common parent for all list forms of documents:

CommonForm
- *BaseListForm*
  - *BaseDocumentEditForm*

On a form the element of management *SplitContainer*, is placed, which keeps the position in settings of the form. The left panel of the container is reserved under placement of the filter (the table or a tree) if to leave this area empty – it will be hidden in the total form.



The form is used for creation of list forms of the documents with screen logic, significantly other than the list form by default. To create a master-detail of the interface on the basis of a list form it is recommended to use *BaseFlatDocumentListForm* by default.

In the class of form *BaseDocumentListForm* *DocumentManager* is imported.

When you deriving from the form *BaseListForm* the following methods and properties of its class can be useful to the application developer:

- *CustomFormText* of type *bool* – returns *true*, if the property *Text* of the form has to remain so as it has been specified at its development. Otherwise the system will create the form heading independently.

It is also possible to redefine the following methods and properties of a class of the *DocumentListForm* form:

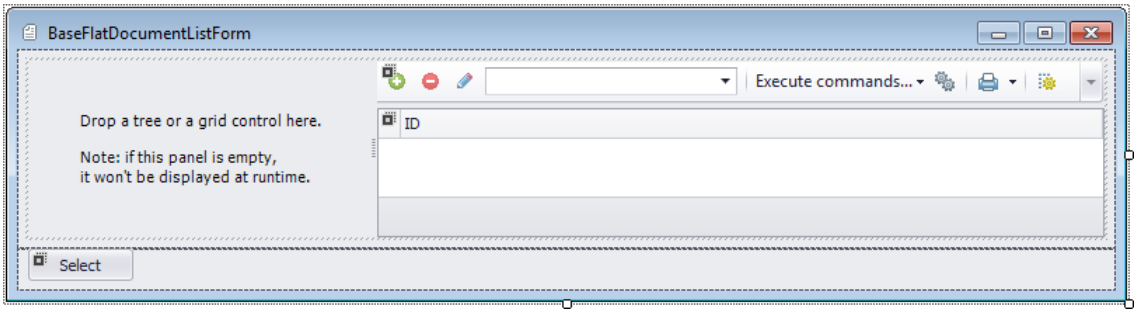
- *DocumentType* of the type *Type* – type of the dictionary which is displayed by the form.

BaseFlatDocumentListForm

BaseFlatDocumentListForm form (from *Ultima.Client.Documents namespace*) is derived from *BaseDocumentListForm* and is common parent for all list forms of documents:

CommonForm
- *BaseListForm*
  - *BaseDocumentListForm*
    - *BaseFlatDocumentListForm*

The most frequent reason to create own list form of the documents is implementation of the master-detail interface. Therefore, *BaseFlatDocumentListForm* form already hosts *SplitContainer control element derived from BaseDocumentListForm* , on which right panel *DocumentGridViewPanel*control element is located,, designed to view the list of documents. The element contains a toolbar with the entire standard functionality - set of columns, filters, etc. Only left panel, which is reserved for location of the filter, is available for use to the application developer. If this area is left empty – it will be hidden in the final form:



To implement own list form of documents, it should be derived from *BaseFlatDocumentListForm* form and *IRecordBrowser<T>* and *IRecordSelector<T>* interfaces should be implemented, where *T* is a document type. The system will search for the form implementing *IRecordBrowser<T>* interface to display the list of documents, and for selection of the documents it will search for the form implementing *IRecordSelector<T>*. If no such form appears to be in the system, the basic list form of documents will open. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator. The process of creation of own documents list form is detailed in the chapter List forms of documents.

The class of *BaseFlatDocumentListForm* form implements the following methods and has the following properties:

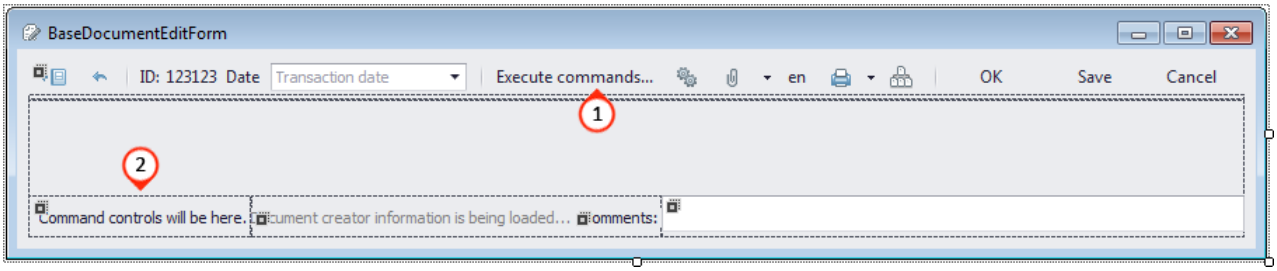
- *DocumentType*, type of *type* returns a document type;
- *SelectedID*, type of *long* returns ID of the document selected in *DocumentGridViewPanel* control element;
- *SelectedList*, type of *IDList* returns a list of IDs of the document selected in *DocumentGridViewPanel* control element;
- *LoadRecords()* loads the documents into *DocumentGridViewPanel* control element;
- *GridPanel*, type of *DocumentGridViewPanel* returns *DocumentGridViewPanel* control element. Using this property, a value can be set for instance for *DocumentGridViewPanel* properties, while customizing the interface of this control element.

BaseDocumentEditForm

The form *BaseDocumentEditForm* (from namespace *Ultima.Client.Documents*) is derived from *BaseEditForm* and is common parent for all edit forms of documents:

CommonForm
- *BaseEditForm*
  - *BaseDocumentEditForm*

The form has implemented support for the commands on a document and list of document links to other documents. The form contains also a toolbar with the functionality enumerated and derived from *BaseEditForm*:



To implement own edit form for a document, it should be derived from *BaseDocumentEditForm* form, and *IRecordEditor<T>* interface should be implemented, where *T* is a document type. The system for editing of the document will search for the form implementing *IRecordEditor<T>* interface. If no such form appears to be in the system, the basic document edit form opens. If more than one of such form appears to be in the system, the system throws an error. It allows avoiding unobvious behaviour of the system in case of error in the system setting by the administrator. The process of creation of own document edit form is detailed in the chapter Edit forms of documents.

The control element has the following specific properties (the properties derived from *BaseEditForm* are described in the corresponding section):

- *CommandsMenuVisible* – if *true*, a button is displayed for the list of commands on a document (1);

- *QuickCommandsVisibles* – if *true*, a toolbar is displayed for quick access to the commands at the form bottom (**2**).

In the class of *BaseDocumentEditForm* form, *DocumentManager* is imported.

When you deriving from the form *BaseDocumentEditForm* the following methods and properties of its class can be useful to the application developer:

- *DocumentType*, type of *type* returns a document type, edited by the current form;
- *Document*, type of *IDocument* – returns the document edited in form – the object *DataRecord*, given to the type *IDocument*.

DictionaryHelper

In the class *DictionaryHelper* (from the namespace *Ultima.Dictionaries*) the methods of the following operations for the dictionaries are realized:

- *EditRecord(Type dictionaryType, long id)* – opens the specified record of the dictionary in a modal form of editing (the information on modal and non-modal forms of editing can be found on the website MSDN eng/rus):
  - *dictionaryType* – dictionary type;
  - *id* – dictionary record id;
- *EditRecord(long id)* – opens the specified record of the dictionary of metadata in a modal form of editing:
  - *id* – metadata object id (dictionary type is uniquely determined by the record code);
- *BeginEditRecord(Type dictionaryType, long id)* – opens the specified record of the dictionary in non-modal form of editing:
  - *dictionaryType* – dictionary type;
  - *id* – dictionary record id;
- *BeginEditRecord(long id)* – opens the specified record of the dictionary in non-modal form of editing:
  - *id* – dictionary record id (dictionary type is uniquely determined by the record code);
- *ViewEditRecord(Type dictionaryType, long id)* – opens the specified record of the dictionary in the reading mode of a modal form of editing:
  - *dictionaryType* – dictionary type;
  - *id* – dictionary record id;
- *BeginViewRecord(Type dictionaryType, long id)* – opens the specified record of the dictionary in the reading mode of non- modal form of editing:
  - *dictionaryType* – dictionary type;
  - *id* – dictionary record id;
- *InsertRecord(Type dictionaryType, IDictionary<string, object> parameters = null)* – opens a modal form of editing for creation of new a record of the dictionary:
  - *dictionaryType* – dictionary type;
  - *parameters* – parameters of a new record of the dictionary (optionally);
- *BeginInsertRecord(Type dictionaryType, IDictionary<string, object> parameters = null)* – opens non-modal form of editing for creation of new a record of the dictionary:
  - *dictionaryType* – dictionary type;
  - *parameters* – parameters of a new record of the dictionary (optionally);
- *BrowseRecords(Type dictionaryType, long? id = null)* – opens non-modal list form of the specified dictionary for viewing of its records:
  - *dictionaryType* – dictionary type;
  - *id* – dictionary record id, which will be chosen in the list (optionally);
- *SelectRecord(Type dictionaryType, long? id = null)* – opens a modal list form of the specified dictionary for choosing of its one record. Returns a code of the chosen record if it was chosen, otherwise *null*:
  - *dictionaryType* – dictionary type;

- *id* – dictionary record id, which will be chosen in the list (optionally);
- *SelectRecords(Type dictionaryType)* – opens a modal list form of the specified dictionary for choosing several records. Returns a list of codes of the chosen records if they were chosen, otherwise *null*:
  - *dictionaryType* – dictionary type.

The typified options of all listed methods are also realized.

DocumentHelper

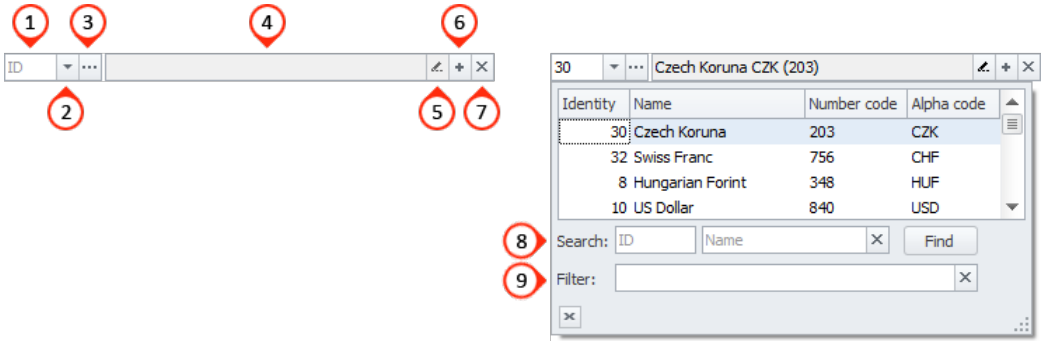
In the class *DocumentHelper* (from namespace *Ultima.Documents*) methods of the following operations on documents are implemented:

- *EditDocument(long id)* opens the specified document in modal edit form (information about modal and modeless edit forms can be found on MSDN website eng/rus):
  - *id* – document ID;
- *BeginEditDocument(long id)* opens the specified document in modeless edit form:
  - *id* – document ID;
- *ViewDocument(long id)* opens the specified document in the reading mode in the modal edit form:
  - *id* – document ID;
- *BeginViewDocument(long id)* opens the specified document in the reading mode in modeless edit form:
  - *id* – document ID;
- *InsertDocument(Type documentType, IDictionary<string, object> parameters = null)* opens the modal edit form for creation of new document:
  - *documentType* – document type;
  - *parameters* – parameters of new document (optional);
- *BeginInsertDocument(Type documentType, IDictionary<string, object> parameters = null)* opens the modeless edit form for creation of new document:
  - *documentType* – document type;
  - *parameters* – parameters of new document (optional);
- *BrowseDocuments(Type documentType, long? id = null)* – opens not model list form to view the documents of specified type:
  - *documentType* – document type;
  - *id* – ID of the document, which will be selected in the list (optional);
- *GetTablePartControl(Type tablePartType)* returns a control element for the table part of specified type:
  - *tablePartType* – table part type;
- *SelectDocument(Type documentType, long? id = null)* – opens model list form to select the document of specified type. It returns ID of selected document, if it was selected, otherwise *null*:
  - *documentType* – document type;
  - *id* – ID of the document, which is selected in the list (optional).

The options of the following methods are implemented too: *InsertDocument<T>*, *BeginInsertDocument<T>*, *BrowseDocuments<T>*, *GetTablePartControl<T>* and *SelectDocument<T>*.

DictionaryLookupEdit

The *DictionaryLookupEdit* control element (from *Ultima.Client.Controls* namespace) performs functions of the *ComboBox* standard control and the *ComboBoxEdit* control of the DevExpress package. Used to select a flat dictionary's record:



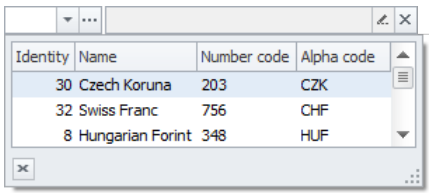
The control has the following properties:

- *AllowClear* – if *true*, the button that resets the selected value is displayed (**7**);
- *DictionaryType* – dictionary that the control will work with;
- *EditButtonEnabled* – if *true*, the button to open the edit form of the selected record (**5**) is enabled, if it is displayed in the control;
- *EditButtonVisible* – if *true*, the button to open the edit form of the selected record is displayed (**5**);
- *FilterPanelVisible* – if *true*, the filter panel is displayed (**9**);
- *IDEditAutoWidth* – if *true* (default value), the field width (**1**) *IDEditWidth* is considered minimal (incl. buttons (**2**) and (**3**)). At the same time, if the ID being entered does not fit into the field (**1**), the field width automatically increases according to length of the number being entered;
- *IDEditVisible* – if *true*, the record code field (**1**) is displayed. If *false*, buttons (**2**) and (**3**) move to the right, beyond field (**4**):

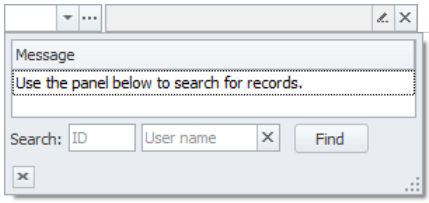
- *IDEditWidth* – total width of the block including field (**1**) and buttons (**2**) and (**3**);
- *LookupButtonVisible* – if *true*, the button to open the control's list is displayed (**2**);
- *NewButtonVisible* – if *true*, the button to open the new record creation form is displayed (**6**);
- *SelectButtonEnabled* – if *true*, the button to open the dictionary list form for selecting a record is enabled (**3**), if it is displayed in the control;
- *SelectButtonVisible* – if *true*, the button to open the dictionary list form for selecting a record is displayed (**3**);
- *ShowColumnHeaders* – if *true*, column headers in the dictionary records list in the control are displayed;
- *Filter* – predicate expression used to filter the list of the available records displayed in the drop-down list or the dictionary list form opened with the select button (**3**).

The functionality and interface of *DictionaryLookupEdit* affected by the following dictionary properties:

- *Is small* – by default, for small dictionaries with the *Is small* flag set the search function is not available. At the same time, the collection of all dictionary records gets into the control when opened;



By default, records from large dictionaries do not get into the control when first opened; To display them, use the search function to shorten the list.



- *Display format* – defines a format for displaying the selected record in the control's field (4). For the currencies dictionary given in the example, the value of this parameter is "`{Name} {AlphaCode} ({NumCode})`". If the *Display format* was not specified, the format will be formulated from the values of all properties of the dictionary, except those of *LargeText* and *byte[]* types;
- *Search property* – defines the property that the search in the dictionary will be performed by. For the currencies dictionary given in the example, the value of this parameter is "`Name`". The filtering is performed by the same parameter;
- *Lookup* – dictionary properties attribute – defines that the dictionary properties will be displayed in a drop-down list of the control (the dictionary record *ID* is always displayed). If neither of dictionary properties is marked with this attribute, the list will include the properties specified in *Display format*. If *Display format* is empty too, all properties will be displayed in the control list, except those of *LargeText* and *byte[]* types;

When creating a new dictionary record, click button (6) can, if needed, define the record's initial values, having responded to *InsertRecord* event. It is possible either to insert parameters for the new record in order that the control handles them singly:

```
private void MyLookupEdit_InsertRecord(object sender, InsertRecordEventArgs args)
{
	args.Parameters["Name"] = "YourText";
}
```

or create the record and return its code to the control in order to use it:

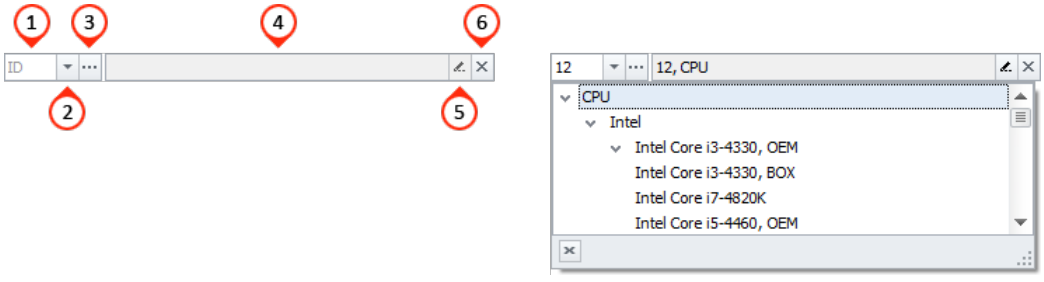
```
private void MyLookupEdit_InsertRecord(object sender, InsertRecordEventArgs args)
{
	args.Handled = true; // important!
	args.InsertedRecordID =
DictionaryHelper.InsertRecord<DictionaryName>(args.Parameters);
}
```

The *RecordCreated* event informs that the record is successfully created.

When editing a dictionary record opened in the control element, one may make use of a similar method by clicking button (5), having responded to *EditSelectedRecord* event.

DictionaryLookupTreeEdit

DictionaryLookupTreeEdit control element (from *Ultima.Client.Controls namespace*) performs the functionality of *ComboBox standard control element* and *ComboBoxEdit*control element of DevExpress package. It is used to select the tree dictionary record:



The control element has the following specific properties:

- *AllowClear* – if *true*, a button is displayed resetting the values selected in the control element (6);
- *DictionaryType* – a dictionary, which data the control element will work with;
- *DisplayColumn* – a name of dictionary property, which is displayed in the control element list tree (the default value is *Name*);
- *EditButtonVisible* – if *true*, a button is displayed to open the edit form for selected record (5);
- *IdColumn* – a name of the dictionary property being ID (the default value is *Id*);
- *IDEditAutoWidth* – if *true*, the field (default) width (1) *IDEditWidth* is interpreted as minimum width of the field (including buttons (2) and (3)). Moreover, if entered ID is not placed in the field (1), the field width is increased automatically in proportion to the sizes of entered number;
- *IDEditVisible* – if *true*a record code field is displayed(1). If *false* buttons (2) and (3) displace to the right after the field (4):

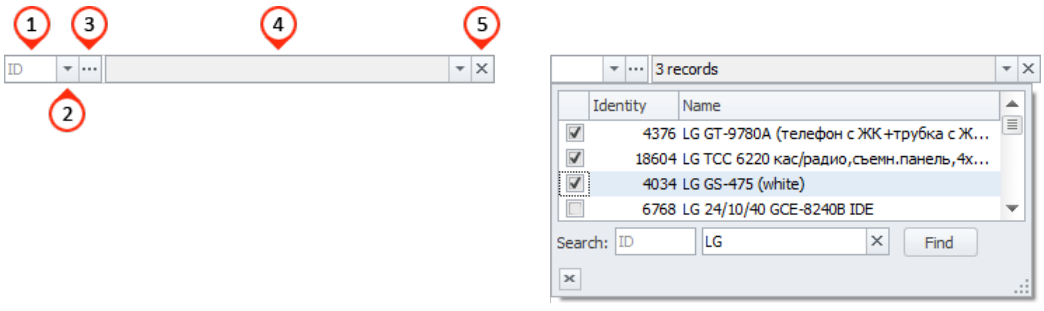
- *IDEditWidth* – overall width of the block, which includes the field (1) and buttons (2), (3);
- *LookupButtonVisible* – if *true*, a button is displayed, by which a list of the control element will pop up(2);
- *ParentColumn* – a name of the dictionary property *Parent property*, according to which a tree is built (the default value is *ParentID*);
- *SelectButtonVisible* – if *true*, a button is displayed for opening of the dictionary list form for selection of the record (3).

The following property of the dictionaries influences the functionality and interface of *DictionaryLookupTreeEdit*:

- *Display format* defines the format, in which the selected record will be displayed in the field (4) of the control element. The value of this parameter for the dictionary of the group of goods given in the example will be "`{ID}, {Name}`". If the value *Display format* is not set, the format will be formulated from the values of all properties of the dictionary, except for the properties of type *LargeText* and *byte[]*.

DictionaryMultiSelectEdit

DictionaryMultiSelectEdit control element (from *Ultima.Client.Controls namespace*) performs the functionality of *CheckedComboBoxEdit* control element of DevExpress package. It is used to select several records of flat and tree dictionary (the records are also provided as a list):



The control element has the following specific properties:

- *DictionaryType* – a dictionary, which data the control element will work with;
- *AllowClear* – if *true* a button is displayed, resetting the values selected in the control element (5);
- *IDEditAutoWidth* – if *true*, the field (default) width (1) *IDEditWidth* is interpreted as minimum width of the field (including buttons (2) and (3)). Moreover, if entered ID is not placed in the field (1), the field width is increased automatically in proportion to the sizes of entered number;

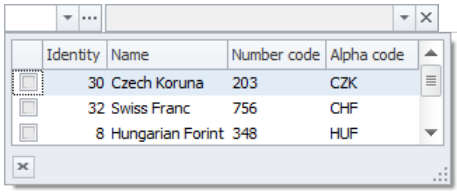
- *IDEditVisible* – if *true*a record code field is displayed(1). If *false* buttons (2) and (3) displace to the right after the field (4):

SCREENSHOT

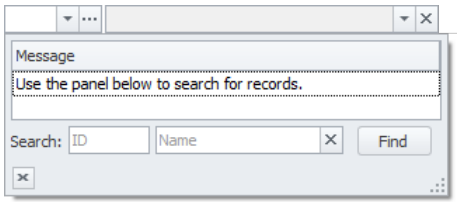
- *IDEditWidth* – overall width of the block, which includes the field (1) and buttons (2), (3);
- *LookupButtonVisible* – if *true*, a button is displayed, by which a list of the control element will pop up(2);
- *SelectButtonVisible* – if *true*, a button is displayed for opening of the dictionary list form for selection of the record (3).

The following properties of the dictionaries influence on the functionality and interface of *DictionaryMultiSelectEdit*:

- *Is small* – for small dictionaries, with set *Is small* flag, the search functionality is unavailable. But at the same time, selection of all dictionary records falls into the control elements when opened;



- The records of large dictionaries do not fall into the control element when opened for the first time. A search limiting the selection should be used for their display;



- *Display format* defines the format, in which the selected record will be displayed in the field (4) of the control element. The value of this parameter for the dictionary of goods given in the example will be "`{ID}, {Name}`". If the value *Display format* is not set, the format will be formulated from the values of all properties of the dictionary, except for the properties of type *LargeText* and *byte[]*:

- *Search property* defines by which dictionary property the search will be carried out. The value of this parameter for the dictionary of goods given in the example will be "`Name`";
- *Lookup* – attribute of dictionary properties defines that the dictionary properties will be displayed in dropdown list of the control element (dictionary record *ID* is displayed always). If neither of dictionary properties is marked with this attribute, the list will include the properties specified in *Display format*. If *Display format* is empty too – all properties will be displayed in the control element list, except for the properties of type *LargeText* and *byte[]*;

DocumentEllipseEdit

The *DocumentEllipseEdit* control (from the *Ultima.Client.Controls* namespace) mimics the functionality of the *ComboBox* standard control and the *ComboBoxEdit* control of the DevExpress package. It is used to select a document of the given type using the document list form:

The control has the following own properties:

- *DocumentType* – document type to be selected using this control.

The control has the following properties inherited from its parent class, *DictionaryLookupEdit*:

- *AllowClear* – if *true*, the button that resets the selected value is displayed (6);
- *DictionaryType* – the value is always set to *typeof(Ultima.Document)*;

- *EditButtonEnabled* – if *true*, the button to open the edit form of the selected record (4) is enabled, if it is displayed in the control;
- *EditButtonVisible* – if *true*, the button to open the edit form of the selected record is displayed (4);
- *IDEditAutoWidth* – if *true* (default value), the field width (1) *IDEditWidth* is considered minimal (incl. button (2)). At the same time, if the ID being entered does not fit into the field (1), the field width automatically increases according to length of the number being entered;
- *IDEditVisible* – if *true*, the record code field (1) is displayed. If *false*, button (2) moves to the right, beyond field (3):

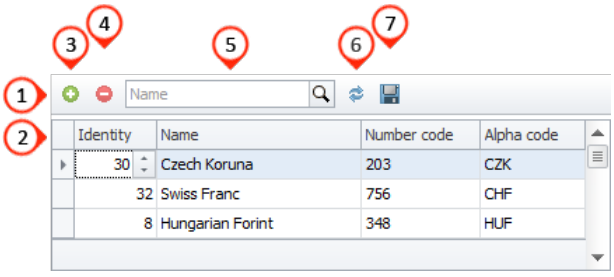
- *IDEditWidth* – total width of the block including field (1) and button (2);
- *LookupButtonVisible* – always *false*, as the drop-down document list is not supported;
- *NewButtonVisible* – if *true*, the button to open the new document creation form is displayed (5);
- *SelectButtonEnabled* – if *true*, the button to open the document list form for selecting a document is enabled (2), if it is displayed in the control;
- *SelectButtonVisible* – if *true*, the button to open the document list form for selecting a document is displayed (2);
- *Filter* – predicate expression used to filter the list of the available records displayed in the drop-down list or the dictionary list form opened with the select button (3). Filter expression can use either the Document type or the concrete type of the document specified in the *DocumentType* property, i.e.:

```
private void InitAgentFilter(long agentId)
{
	SaleDocumentBox.Filter = GetFilter<SaleDocument>(d => d.AgentID == agentId);
}
```

DictionaryGridPanel

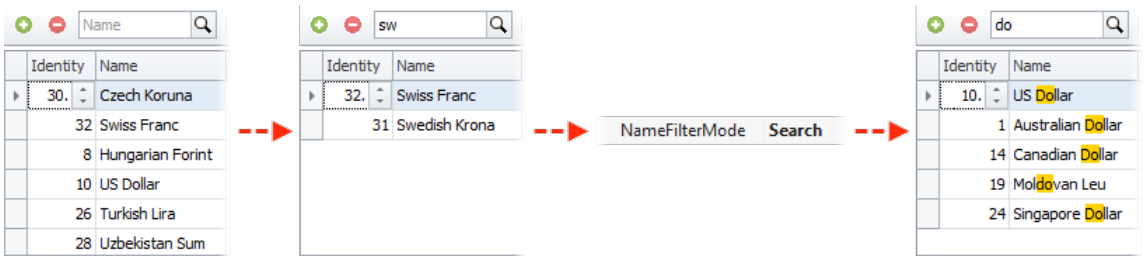
*DictionaryGridPanel*control element (from *Ultima.Client.Controls namespace*) is used to display and edit the content of flat and tree dictionary (the records are also provided in the form of a list). It consists of:

- toolbar (1);
- table of dictionary records (2) – *GridControl* control element of DevExpress package. Editing of records is carried out directly in the table:



The control element has the following specific properties:

- *DictionaryType* – a dictionary, which data the control element works with;
- *AutoPopulateGridColumns* – if *true*, the table columns (*GridControl*) (2) are created automatically for all properties of the dictionary bound to the element. In the value *false*, the columns are used being created by the application developer;
- *CanDragToolbar* – if *true*, it allows changing the position of the toolbar (1);
- *DeleteButtonEnabled* – if *true*, a button for deletion of dictionary records (4), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *DeleteButtonVisible* – if *true*, a button for deletion of dictionary record is displayed (4);
- *NameFilterMode* sets the mode for performance of search using the field (5):
  - if *filter*, (by default) all dictionary records are filtered by entry of searched text;
  - if *search*, a search is carried out only among the records displayed in the table (*GridControl*), no database access is carried out at that. As a result of search, only the records will remain in the table, which meet the condition. Moreover, the entries of searched fragment will be highlighted in them:



- *NameFilterVisible* – if *true*, a search field is displayed (5). The search is carried out by the field of the dictionary, defined with its *Search property*;
- *NameFilterWidth* – width of the search field in pixels (5);
- *NewButtonEnabled* – if *true*, a button for creation of new dictionary record (3), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *NewButtonVisible* – if *true*, a button for creation of new dictionary record is displayed (3);
- *ReloadButtonVisible* – if *true*, a button to reload dictionary record is displayed (6);
- *SaveButtonEnabled* – if *true*, a button for saving of changes made to the dictionary record (7), if displayed in the toolbar, it can be clicked;
- *SaveButtonVisible* – if *true*, a button for saving of changes made to the dictionary record is displayed (7);
- *ToolbarVisible* – if *true*, a toolbar is displayed (1).

While using the control element, in addition to indication of *DictionaryType* , the dictionary of the same type should be bound in its properties to *GridControl* control element, which is its part.

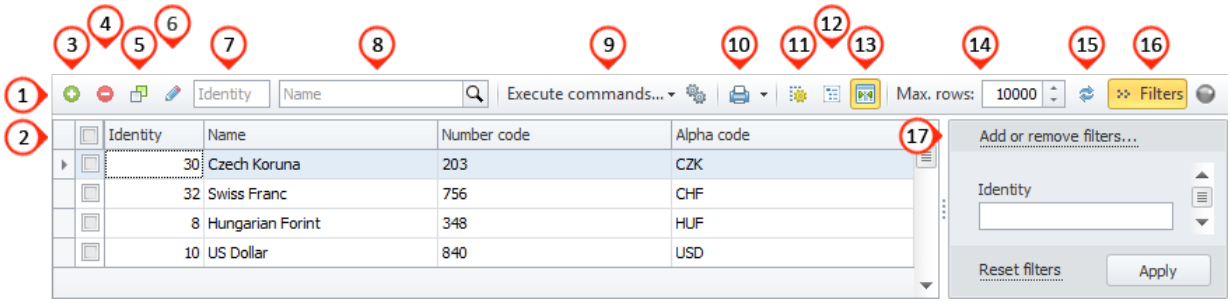
The class of *DictionaryGridPanel* control element implements the following methods and has the following properties:

- *DictionaryType*, type *type* returns a dictionary type;
- *LoadRecords()* loads the dictionary record into *GridControl* control element;
- *SaveRecords()* saves edited dictionary records;
- *SaveAndReload()* saves edited dictionary records and reloads the list of dictionaries;
- *ApplyCustomFilter* – an event, which is executed after loading of records (*LoadRecords*). It supports strictly typed filters and can be executed in asynchronous manner.

DictionaryGridViewPanel

Control element *DictionaryGridViewPanel* (from namespace *Ultima.Client.Controls*) control element is used to display content of flat dictionary. It consists of:

- toolbar (1);
- table of dictionary records (2) – *GridControl* control element of DevExpress package:

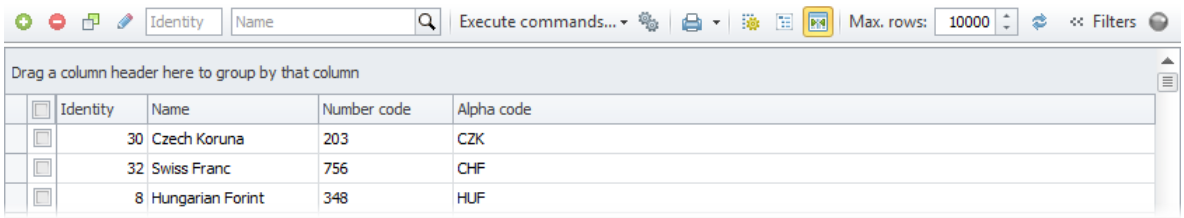


The control element has the following specific properties:

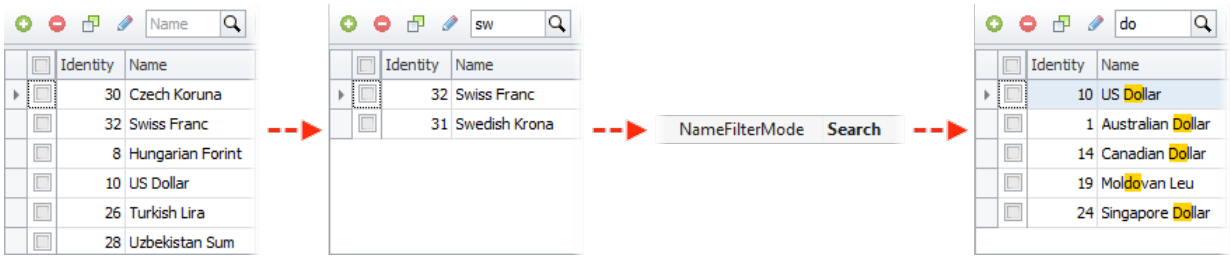
- *DictionaryType* – a dictionary, which data the control element works with;
- *AllowEmptyFilter* – if *true*, it allows applying unfilled filter to dictionary records (17), which will return all dictionary records. If*false* unfilled filter will return *null*;
- *AutoBestFitAfterColumnSelection* – if *true* after selection of columns displayed in the table (*GridControl*), an optimum width will be selected for them;
- *CanDragToolbar* – if *true*, it allows changing the position of the toolbar (1);
- *CheckColumnVisible* – if *true*, a column is displayed to select the dictionary records:

- *CloneButtonEnabled* – if *true*, a button to clone dictionary records (5), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *CloneButtonVisible* – if *true*, a button to clone dictionary record is displayed (5);
- *CommandsMenuVisible* – if *true*, a button is displayed for the list of commands on dictionary records (9). The button is visible only if there are commands on dictionary records for that type;
- *DeleteButtonEnabled* – if *true*, a button for deletion of dictionary records (4), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *DeleteButtonVisible* – if *true*, a button for deletion of dictionary record is displayed (4);
- *EditButtonEnabled* – if *true*, a button to edit dictionary records (6), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *EditButtonVisible* – if *true*, a button to edit dictionary record is displayed (6);
- *FilterButtonVisible* – if *true*, a button to open/hide the filter panel is displayed (16);
- *FilterPanelVisible* – if *true*, a filter panel is displayed (17);
- *FilterPanelWidth* – filter panel width in pixels (17) by default;
- *FitColumnButtonVisible* – if *true*, a button for automatic selection of optimum width of the table column is displayed (*GridControl*) (13);

- *GroupButtonVisible* – if *true*, a button to open/hide the grouping panel is displayed (**12**);
- *GroupPanelVisible* – if *true*, a grouping panel is displayed under the toolbar (**1**):



- *IDEditVisible* – if *true*, a field is displayed for quick opening of the record according to the code (**7**):
- *LimitCounterValue* – the default value of the counter (**14**), limiting the number of dictionary records displayed in the form;
- *LimitCounterVisible* – if *true*, a counter is displayed (**14**), which limits the number of dictionary records displayed in the form;
- *LimitCounterWidth* – width of counter value input in pixels (**14**);
- *NameFilterMode* sets the mode for performance of search using the field (**8**):
  - if *filter*, (by default) all dictionary records are filtered by entry of searched text;
  - if *search*, a search is carried out only among the records displayed in the table (*GridControl*), no database access is carried out at that. As a result of search, only the records will remain in the table, which meet the condition. Moreover, the entries of searched fragment will be highlighted in them:



- *NameFilterVisible* – if *true*, a search field is displayed (**8**). The search is carried out by the field of the dictionary, defined with its *Search property*;
- *NameFilterWidth* – width of the search field in pixels (**8**);
- *NewButtonEnabled* – if *true*, a button for creation of new dictionary record (**3**), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *NewButtonVisible* – if *true*, a button for creation of new dictionary record is displayed (**3**);
- *PrintButtonVisible* – if *true*, a print button is displayed (**10**);
- *QuickCommandsVisible* – if *true,* a panel with commands on dictionary records, added to quick access, is displayed at the bottom of control element. The panel is visible only if the commands were added to it;
- *ReloadButtonVisible* – if *true*, a button to reload dictionary record is displayed (**15**);
- *SaveSelectedColumns* – if *true*, the selection of buttons, performed by the user using corresponding tool (**11**), will be remembered;
- *SelectColumnsButtonEnabled* – if *true*, a button for selection of columns (**11**), if displayed in the toolbar, it can be clicked.
- *SelectColumnsButtonVisible* – if *true*, a button is displayed for selection of columns visible in the table – properties of dictionary records (**11**);
- *ToolbarVisible* – if *true*, a toolbar is displayed (**1**).

While using the control element, in addition to indication of *DictionaryType* , the dictionary of the same type should be bound in its properties to *GridControl* control element, which is its part (the process is detailed in the chapter List forms of dictionaries).

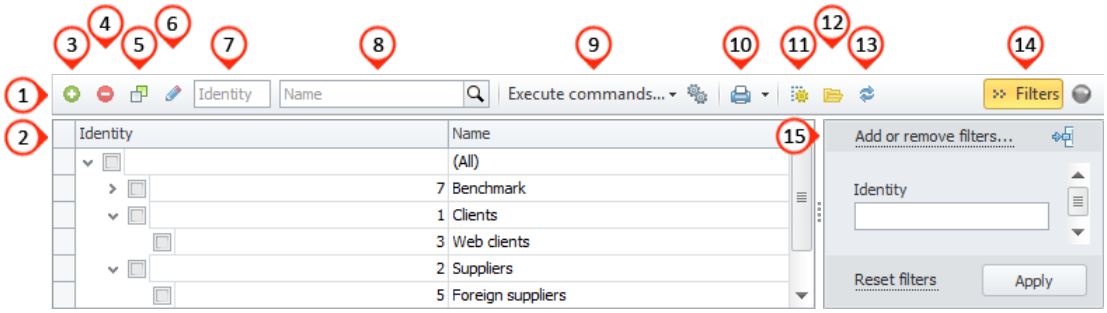
The class of *DictionaryGridViewPanel* control element implements the following methods and has the following properties:

- *DictionaryType*, type *type* returns a dictionary type;
- *CheckedRecords*, type *long[]* returns an array of IDs of dictionary records selected in the control element;
- *LoadRecords()* loads the dictionary record into *GridControl* control element;
- *ApplyCustomFilter* – an event, which is executed after loading of records (*LoadRecords*). It supports strictly typed filters and can be executed in asynchronous manner.

DictionaryTreeViewPanel

The *DictionaryTreeViewPanel* control element (from *Ultima.Client.Controls* namespace) is used to display content of tree dictionary. It consists of:

- toolbar (1);
- table of dictionary records (2) – *TreeList* control element of DevExpress package:

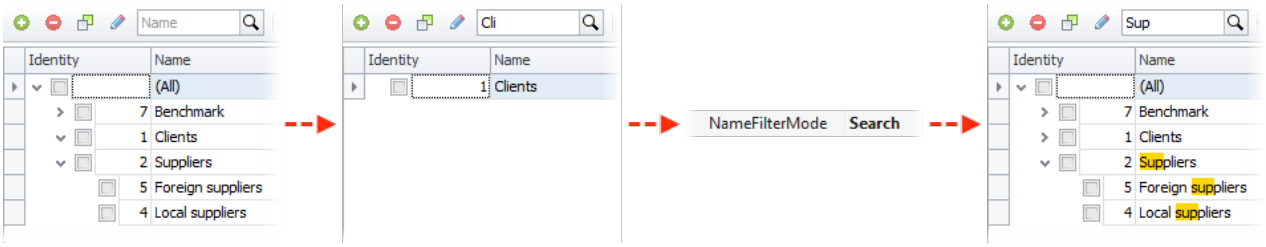


The control element has the following specific properties:

- *DictionaryType* – a dictionary containing data that the control element works with;
- *AllowEmptyFilter* – if *true*, allows applying unfilled filter to dictionary records (15), which will return all dictionary records. if *false*, unfilled filter will return *null*;
- *AutoBestFitAfterColumnSelection* – if *true* after selection of columns displayed in the tree (*TreeList*), an optimum width will be selected for them;
- *CanDragToolbar* – if *true*, it allows changing the position of the toolbar (1);
- *CheckColumnVisible* – if *true*, a column is displayed to select the dictionary records:

- *CloneButtonEnabled* – if *true*, a button to clone dictionary records (5), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *CloneButtonVisible* – if *true*, a button to clone dictionary record is displayed (5);
- *CommandsMenuVisible* – if *true*, a button is displayed for the list of commands on dictionary records (9). The button is visible only if there are commands on dictionary records for that type;
- *CommonRootNodeVisible* – if *true* , the level, which is parent for all dictionary records, is displayed *(All)* in the tree (*TreeList*);
- *DeleteButtonEnabled* – if *true*, a button for deletion of dictionary records (4), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *DeleteButtonVisible* – if *true*, a button for deletion of dictionary record is displayed (4);

- *EditButtonEnabled* – if *true*, a button to edit dictionary records (6), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *EditButtonVisible* – if *true*, a button to edit dictionary record is displayed (6);
- *FilterButtonVisible* – if *true*, a button to open/hide the filter panel is displayed (14);
- *FilterPanelVisible* – if *true*, a filter panel is displayed (15);
- *FilterPanelWidth* – filter panel width in pixels (15) by default;
- *IDEditVisible* – if *true*, a field is displayed for quick opening of the record according to the code (7):
- *NameFilterMode* sets the mode for performance of search using the field (8):
  - if *filter*, (by default) all dictionary records are filtered by entry of searched text;
  - if *search*, a search is carried out only among the records displayed in the tree (*TreeList*), no database access is carried out at that. As a result of search, all previously displayed records will remain in the table, but at the same time, the entries of searched fragment will be highlighted:



- *NameFilterVisible* – if *true*, a search field is displayed (8). The search is carried out by the field of the dictionary, defined with its *Search property*;
- *NameFilterWidth* – width of the search field in pixels (8);
- *NewButtonEnabled* – if *true*, a button for creation of new dictionary record (3), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary form;
- *NewButtonVisible* – if *true*, a button for creation of new dictionary record is displayed (3);
- *PrintButtonVisible* – if *true*, a print button is displayed (10);
- *QuickCommandsVisible* – if *true,* a panel with commands on dictionary records, added to quick access, is displayed at the bottom of control element. The panel is visible only if the commands were added to it;
- *ReloadButtonVisible* – if *true*, a button for reload of dictionary record is displayed (13);
- *SaveSelectedColumns* – if *true*, the selection of buttons, performed by the user using corresponding tool (11), will be remembered;
- *SelectColumnsButtonEnabled* – if *true*, a button for selection of columns (11), if displayed in the toolbar, it can be clicked.
- *SelectColumnsButtonVisible* – if *true*, a button is displayed for selection of columns visible in the table – properties of dictionary records (11);
- *SubfoldersButtonChecked* – if *true*, the subfolder button (12) is clicked;
- *SubfoldersButtonVisible* – if *true*, a subfolder button is displayed (12);
- *ToolbarVisible* – if *true*, a toolbar is displayed (1).

While using the control element, in addition to indication of *DictionaryType* in its properties, the dictionary of the same type should be bound to *TreeList* control element, which is its part.

The class of *DictionaryTreeViewPanel* control element implements the following methods and has the following properties:

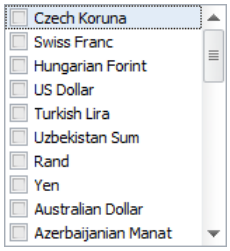
- *DictionaryType*, type *type* returns a dictionary type;
- *SelectedList*, type *IDList* returns a list of IDs of dictionary records selected in the control element;
- *GetIDList(IEnumerable<TreeListNode> nodes)* returns a list of IDs of dictionary records for specified nodes of the tree;
- *LoadRecords()* loads the dictionary record into *TreeList* control element;

- *ApplyCustomFilter* – an event, which is executed after loading of records (*LoadRecords*). It supports strictly typed filters and can be executed in asynchronous manner.

DictionaryCheckList

Control element *DictionaryChekList* (from namespace *Ultima.Client.Controls*) is used to select the dictionary records.

The control element has the following specific properties:



- *DictionaryType* – a dictionary, which data the control element works with;
- *PropertyName* – a name of dictionary property, which values will be displayed in the control element;
- *SaveCheckedRecords* – if *true*, the marked dictionary records are saved in user settings of the form and are selected automatically when re-opened.

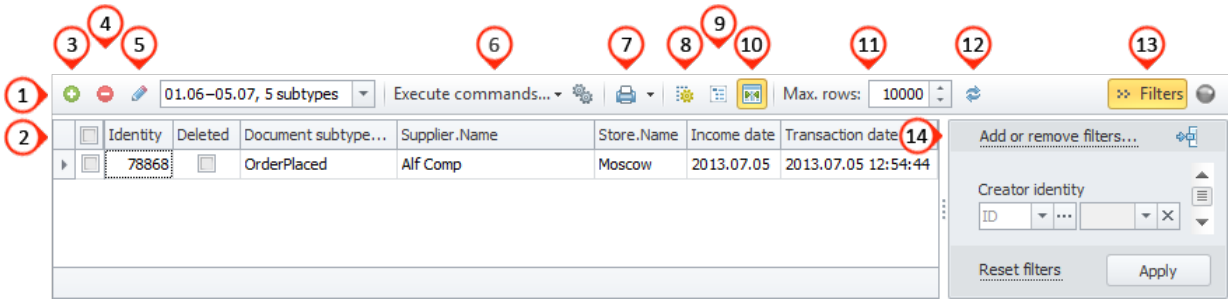
The class of control element implements the following methods and has the following properties:

- *CheckedRecords*, type *IDList* returns a list of IDs of dictionary records, marked with flags in the control element, or sets flags for the list of assigned IDs;
- *CheckAll()* checks all dictionary records, loaded into control element;
- *CheckNone()* unchecks all dictionary records in the control element;
- *CheckedRecordsChanged* – event fired when selection of records is changed.

DocumentGridViewPanel

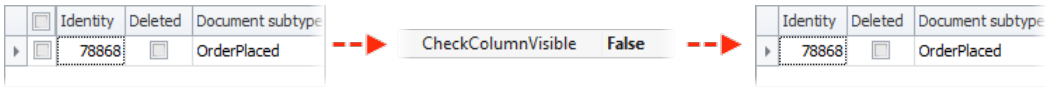
The *DocumentGridViewPanel* control element (from *Ultima.Client.Controls* namespace) is used to display content of the logs of documents. It consists of:

- toolbar (1);
- tables containing documents (2) – *GridControl* control element of DevExpress package:



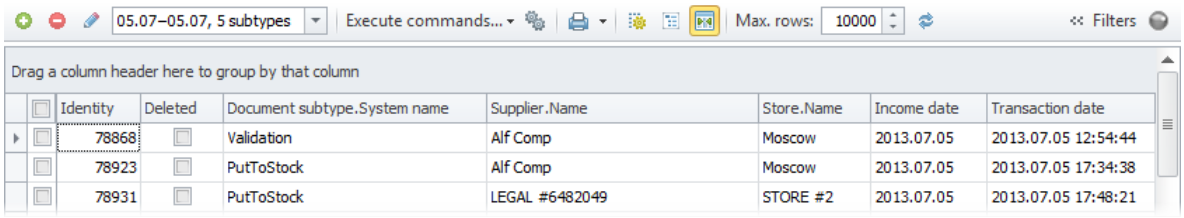
The control element has the following specific properties:

- *DocumentType* – a type of document, which the control element works with;
- *AutoBestFitAfterColumnSelection* – if *true* after selection of columns displayed in the table (*GridControl*), an optimum width will be selected for them;
- *CanDragToolbar* – if *true*, it allows changing the position of the toolbar (1);
- *CheckColumnVisible* – if *true*, a column is displayed to select the documents:



- *CommandsMenuVisible* – if *true*, a button is displayed for the list of commands on documents (6). The button is visible only if there are commands on documents for that type;

- *DeleteButtonEnabled* – if *true*, a button for deletion of a document (4), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens a log of documents;
- *DeleteButtonVisible* – if *true*, a button to delete a document is displayed (4);
- *EditButtonEnabled* – if *true*, a button to edit a document (5), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens a log of documents;
- *EditButtonVisible* – if *true*, a button to edit a document is displayed (5);
- *FilterButtonVisible* – if *true*, a button for opening/hiding of the filter panel is displayed (13);
- *FilterPanelVisible* – if *true*, a filter panel is displayed (13);
- *FilterPanelWidth* – filter panel width in pixels (13) by default;
- *FitColumnButtonVisible* – if *true*, a button for automatic selection of optimum width of the table column is displayed (*GridControl*) (10);
- *GroupButtonVisible* – if *true*, a button to open/hide the grouping panel is displayed (9);
- *GroupPanelVisible* – if *true*, a grouping panel is displayed under the toolbar (1):



- *LimitCounterValue* – the default value of the counter (11), limiting the number of documents displayed in the form;
- *LimitCounterVisible* – if *true*, a counter is displayed (11), which limits the number of documents displayed in the form;
- *LimitCounterWidth* – width of counter value input in pixels (11);
- *NewButtonEnabled* – if *true*, a button to create a new document (3), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens a log of documents;
- *NewButtonVisible* – if *true*, a button to create a new document is displayed (3);
- *PrintButtonVisible* – if *true*, a print button is displayed (7);
- *QuickCommandsVisible* – if *true,* a panel with commands on documents, added to quick access, is displayed at the bottom of control element. The panel is visible only if the commands were added to it;
- *ReloadButtonVisible* – if *true*, a button for reload of documents is displayed (12);
- *SaveSelectedColumns* – if *true*, the selection of buttons, performed by the user using corresponding tool (8), will be remembered;
- *SelectColumnsButtonEnabled* – if *true*, a button for columns selection (8), if displayed in the toolbar, it can be clicked.
- *SelectColumnsButtonVisible* – if *true*, a button is displayed for selection of columns visible in the table – properties of a document (8);
- *ToolbarVisible* – if *true*, a toolbar is displayed (1).

While using the control element, in addition to indication of document type *DictionaryType* , the document of the same type should be bound in its properties to *GridControl* control element, which is its part.

The class of *DocumentGridViewPanel* control element implements the following methods and has the following properties:

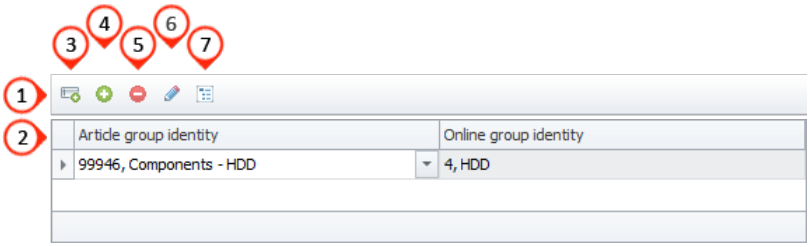
- *DocumentType*, type of *type* returns a document type;
- *CheckedRecordsCount*, type of *int* returns a number of documents selected in the control element;
- *CheckedRecords*, type of *long[]* returns an array of IDs of documents selected in the control element;

- *LoadRecords()* loads the dictionary record into *GridControl* control element;
- *ApplyCustomFilter* – an event, which is executed after loading of records (*LoadRecords*). It supports strictly typed filters and can be executed in asynchronous manner.

LinkTableGridPanel

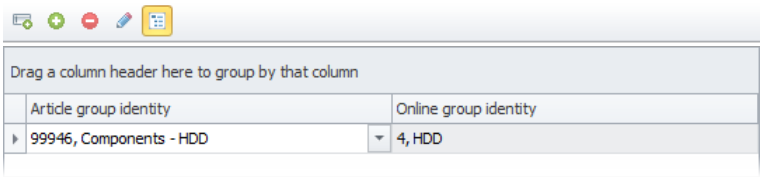
LinkTableGridPanel control element (from *Ultima.Client.Controls namespace*) is used to display and edit the content of link tables in the edit form of dictionary records. It consists of:

- toolbar (1);
- table of link table records (2) – *GridControl* control element of DevExpress package. Editing of records can be carried out directly in the table:



The control element has the following specific properties:

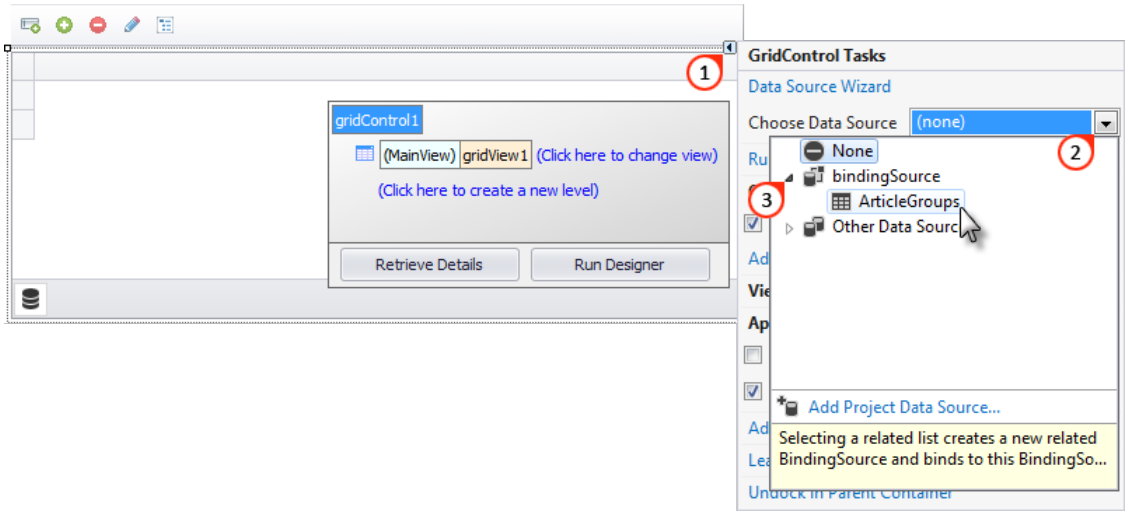
- *LinkTableType* – a link table, which data the control element works with;
- *LinkTableEditableReference* – a name of the dictionary, which the link table refers to in the format *Name* of the dictionary. If the name is set, upon click on the button (4) a form opens for editing of specified dictionary to create a new record. After creation, this record is added to the link table;
- *AutoPopulateGridColumns* – in the value *true*, the table columns (*GridControl*) (2) are created automatically for all properties of the link table bound to the control element. In the value *false*, the columns are used being created by the application developer;
- *CanDragToolbar* – if *true*, it allows changing the position of the toolbar (1);
- *DeleteButtonEnabled* – if *true*, a button to delete link table records (5), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary record edit form;
- *DeleteButtonVisible* – if *true*, a button to delete link table record is displayed (5);
- *EditButtonEnabled* – if *true*, a button to edit link table records (6), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary record edit form;
- *EditButtonVisible* – if *true*, a button to edit link table record is displayed (6);
- *GroupButtonVisible* – if *true*, a button to open/hide the grouping panel is displayed (7);
- *GroupPanelVisible* – if *true*, a grouping panel is displayed under the toolbar (1):



- *NewButtonEnabled* – if *true*, a button to create a new link table record (4), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary record edit form;
- *NewButtonVisible* – if *true*, a button to create a new link table record is displayed (4);
- *NewInlineButtonEnabled* – if *true*, a button to create new empty row in the table (*GridControl*) of the control element (3), if displayed in the toolbar, it can be clicked. The property is set automatically depending on the permissions of the user that opens the dictionary record edit form;

- *NewInlineButtonVisible* – if *true*, a button is displayed to create new empty row in the table (*GridControl*) of the control element (**3**);
- *ToolbarVisible* – if *true*, a toolbar is displayed (**1**).

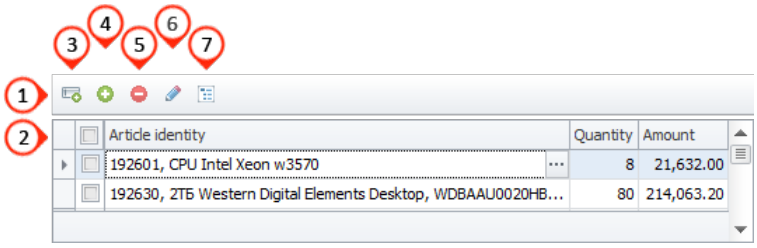
While using the control element, in addition to indication of *LinkTableType* in its properties, the link table of the same type should be bound to *GridControl* control element, which is its part. Moreover, it should be executed through the data source of dictionary edit form, in which a link table should be selected by the name of the second dictionary, which it refers to:



BaseTablePartGridPanel

The *BaseTablePartGridPanel* control element (from *Ultima.Client.Controls* namespace) is used to display and edit the content of table parts in the document edit form. It consists of:

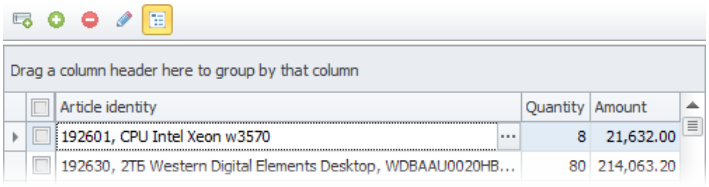
- toolbar (**1**);
- table of table part records (**2**) – *GridControl* control element of DevExpress package. Editing of records can be carried out directly in the table:



The control element has the following specific properties:

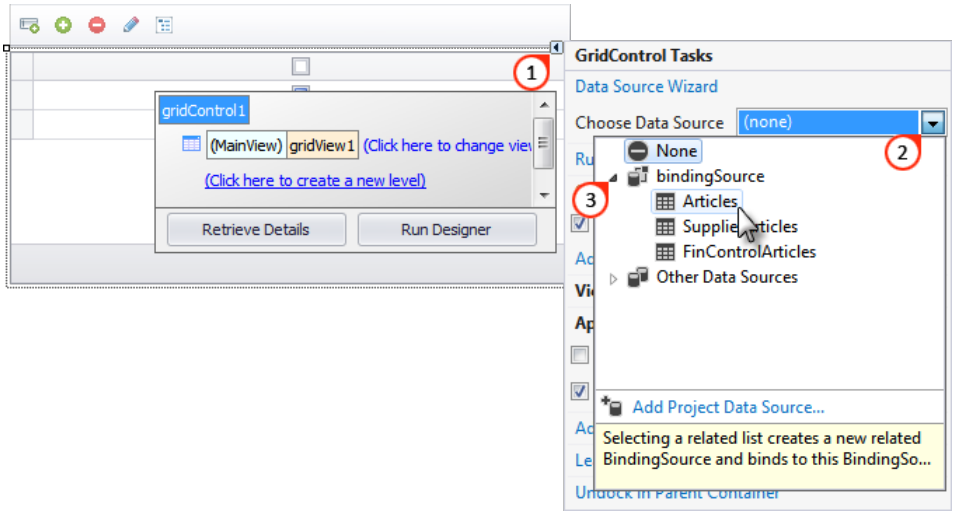
- *TablePartType* – table part containing data that the control element works with;
- *AutoPopulateGridColumns* – if *true*, the table columns (*GridControl*) (**2**) are created automatically for all properties of the table part bound to the control element. If *false*, the columns created by the application developer are used;
- *CanDragToolbar* – if *true*, it allows changing the position of the toolbar (**1**);
- *DeleteButtonEnabled* – if *true*, the button to delete table part records (**5**), if displayed in the toolbar, can be clicked. The property is set automatically depending on the permissions of the user that opens the document edit form;
- *DeleteButtonVisible* – if *true*, the button to delete table part records is displayed (**5**);
- *EditButtonEnabled* – if *true*, the button to edit table part records (**6**), if displayed in the toolbar, can be clicked. The property is set automatically depending on the permissions of the user that opens the document edit form;
- *EditButtonVisible* – if *true*, the button to edit table part records is displayed (**6**);

- *GroupButtonVisible* – if *true*, the button to open/hide the grouping panel is displayed (7);
- *GroupPanelVisible* – if *true*, the grouping panel is displayed under the toolbar (1):



- *NewButtonEnabled* – if *true*, the button for creation of a new table part record (4), if displayed in the toolbar, can be clicked. The property is set automatically depending on the permissions of the user that opens the document edit form;
- *NewButtonVisible* – if *true*, the button for creation of a new table part record is displayed (4);
- *NewInlineButtonEnabled* – if *true*, the button to create a new empty row in the table (*GridControl*) of the control element (3), if displayed in the toolbar, can be clicked. The property is set automatically depending on the permissions of the user that opens the document edit form;
- *NewInlineButtonVisible* – if *true*, the button is displayed to create a new empty row in the table (*GridControl*) of the control element (3);
- *ToolbarVisible* – if *true*, a toolbar is displayed (1).

When using the control element, in addition to indication of *TablePartType* in its properties, the table part of the same type should be bound to *GridControl* control element, which is its part. Moreover, it should be executed through the data source of document edit form, in which a corresponding table part should be selected:

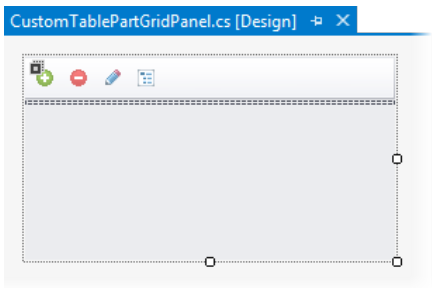


The class of *BaseTablePartGridPanel* control element has the following properties:

- *TablePartType*, type *type* returns a table part type;
- *Selection*, type *IDList* returns a list of IDs of table part rows selected in the control element.

The *BaseTablePartGridPanel* control element can be both placed directly on the document edit form and own control element of the table part can be implemented on its basis. The second method is more preferable when the functionality of basic document edit form is sufficient in general and an upgrade is required only for the table part.

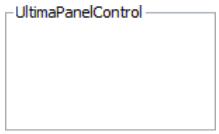
The most frequent reasonto create own control element of the table part is implementation of any additional non-standard logic. Therefore, only the toolbar will be located in own control element derived from *BaseFlatDictionaryListForm* (1). The application developer must locate the table (Grid) to display the table part records independently (2), e.g., *GridControl* control element of DevExpress package.



To implement own control element of the table part, it should be derived from *BaseTablePartGridPanel* control element and *ITablePartEditor<T>* interface should be implemented, where *T* is a type of the table part. The system will search for the control element implementing *ITablePartEditor<T>* interface to display the table part. If no such control element form appears to be in the system, the basic document table part will open. If more than one of such control element appears to be in the system, the system will throw an error. It allows avoiding non-obvious behaviour of the system in case of error in the system setting by the administrator. The detailed information on creation of a dictionary's customized list form provided in the chapter Table parts.

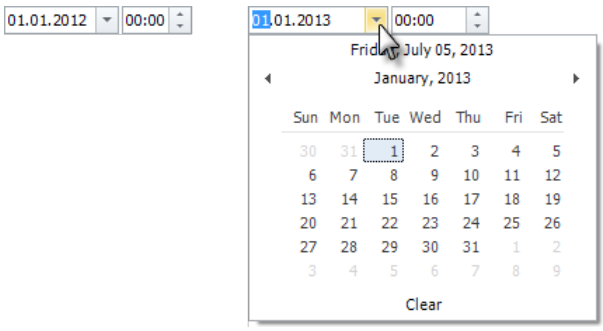
UltimaPanelControl

Control element *UltimaPanelControl* (from namespace *Ultima.Client.Controls*) executes the functionality of *PanelControl* and standard control element *GroupControl* of DevExpress package. But as distinguished from them, *UltimaPanelControl* changes its interface depending on the used topic and has transparent background. It is used to unite the control elements into a group.



UltimaDateEdit

UltimaDateEdit control element executes the functionality of *DateTimePicker* standard control element and *DateEdit* control element of DevExpress package. It is used for input of data and time:



The control element has the following specific properties:

- *DateTime* – default value. If the property is not defined, the control element will open empty:

- *TimeEditVisible* – if *true*, a time block (on the right) will be displayed. It is used to handle with data types *date* and *DateTime*;
- *TimeEditWidth* – time block width.

The control element class has the following properties:

- *DateTime*, type *DateTime* returns time entered in the control element;
- *EditValueChanged* – an event fired in case of change of value in the control element.

UltimaFileEdit

Control element *UltimaFileEdit* (from namespace *Ultima.Client.Controls*) is used to load and save files.

The control element class has the following properties:

- *FileName*, type *string* returns a file name;
- *FileData*, type *byte[]* returns a file;
- *CheckNone()* unchecks all dictionary records in the control element.
- *FileRead* – an event fired when a file is read from the disk (at the point of confirmation of file selection);
- *FileNameChanged* – an event fired when the file name is changed;
- *FileDataChanged* – an event fired when the file is changed.

UltimaTextEdit

Control element *UltimaTextEdit* (from namespace *Ultima.Client.Controls*) executes the functionality of standard control elements *TextBox and RichTextBox*, as well as control element *TextEdit* of DevExpress package. It is used for input of text:

The control element has the following specific properties:

- *MaxStringSize* – maximum size of the string (text), the counter functioning is based on this parameter (**1**). The default value is 256;
- *Multiline* – if *true*, the input field is transformed from the row into a text block. In addition to difference in interface, a symbol of row break is perceived in such control element;
- *PropertyID* – ID of multilanguage property of the dictionary (marked with *Multilanguage* flag), which data the control element will handle;
- *ReadOnly* – in value *true* the text input manually in the element of the control will be forbidden;
- *ShowCounter* – in value *true* the counter of quantity of symbols, available to input (**1**). In value *false* restrictions input by parameter *Max size* dictionary properties, are not applied;
- *ShowTranslationButton* – in value *true* the button of input of multilingual values of dictionary properties is displayed (**2**).

PostgreSQL-based version features

PostgreSQL version limitations

Compared to the Oracle-based version, there are following limitations:

- Predicates aren't supported.
- Values of the DateTime type are not automatically translated in accordance to the user's time zone.
- Application server call and session data (ClientInfo) is not displayed in the system views (pg_stat_activity).
- Automatic Oracle session sweeper task is disabled.

- Multiple active DataReaders sharing a database connection (also called MARS — Multiple Active Result Sets) is not allowed.
- Any database error invalidates the current transaction so that it refuses any further SQL commands and cannot be committed.

The rest of the system is implemented on par with the Oracle-based version.

PostgreSQL development features

Here is a brief overview of the distinctive PostgreSQL features that affect us the most. It's by no means comprehensive, but should be considered as an absolute minimum for the Oracle developer:

- Table names, column names and function names are lowercase (except for the double-quoted identifiers). This affects the result set column names of the SQL queries.
- Packages are not supported. Ultimate AEGIS® kernel use schemas to group functions instead of packages. That's why PostgreSQL version uses more kernel schemas.
- Session variables are used instead of the package variables. The semantic is close to package variables, but not fully equivalent.
- Temporary tables in PostgreSQL are always local and cannot be bound to specific schemas, unlike Oracle. Ultimate AEGIS® has a set of function providing the emulation layer for Oracle-style global temporary tables that fully supports the familiar syntax of using these tables.
- Triggers don't have their own executable body. Every trigger wraps a call to a stored procedure.
- Deferrable and deferred check constraints aren't supported. They can be emulated using deferrable/deferred triggers.
- Stored procedure syntax is PL/PgSQL which resembles but still is different from Oracle's PL/SQL in many aspects.
- Stored procedures always use dynamic binding (i.e. symbols are bound to their semantic at runtime). For example, when a stored procedure invokes a query such as "SELECT * FROM USERS", the name "USERS" is searched for at runtime, not at compile-time. Dynamic binding feature makes PostgreSQL very flexible, but also vulnerable to the runtime errors, as compared to Oracle.
- By default, PostgreSQL functions use the current database user's permissions (in Oracle, they use function owner's permissions). This setting can be overridden per-function.
- Empty strings are not same as NULLs. Unlike Oracle, concatenating a PostgreSQL varchar or text value with NULL always yields NULL. Always initialize the local variables with an empty string to avoid the unintentional nullification of the result.
- DDL operations in PostgreSQL are transactional. Creating functions, altering column types, truncating tables, etc — need to be committed.
- Replacing a function with a new version fails if the signature is different. In that case, "CREATE OR REPLACE FUNCTION" is not enough: one needs to DROP the old version before creating a new one.
- Any database error invalidates the current transaction so that it refuses any further SQL commands and cannot be committed. But the transaction can be rolled back to the last savepoint before the error, if any. After rolling back to the savepoint, the transaction can be continued and committed (this approach is used by the integration tests).

- Multiple active DataReaders sharing a database connection (also called MARS — Multiple Active Result Sets) is not allowed. To read several data sets in parallel, we either open a separate connection for each data set, or process them sequentially.

The database specifics cannot always be abstracted away in the application code. Many commands or services need to invoke pure dynamic SQL statements, execute stored procedures, etc. Application schema may define custom views, constraints and triggers, so Oracle application developer is forced to learn at least basic PostgreSQL features. The following instructions may be helpful for dealing with a few of the above features.

Working around the dynamic binding in PL/PgSQL code

Dynamic binding is a powerful feature that in some cases can help avoiding the dynamic execution of statements (EXECUTE sql). But on the other hand, dynamic binding loosens the compile-time validity checks. The compiler cannot check whether the given symbol represents an object statically. When a function references a symbol such as table name or a function, the object it references to will be determined at runtime. Also, the runtime search is affected by the built-in "search_path" session variable, which means that the symbol can be bound to any object in any schema according to the user preferences. Of course, this is often not what was intended.

To disable the dynamic binding, we use two rules:

- Add "set search_path to (current schema name)" clause to all functions and
- Qualify all tables and functions outside of the current schema with their schema names.

This doesn't make the bindings static (PostgreSQL still doesn't check the validity of the symbols), but effectively disables the dynamic binding. Here is the example source code for the PL/PgSQL function not affected by the dynamic binding:

```
-- current search_path = my_schema
create or replace function my_func(my_arg text) returns void as $$
declare
       v_id bigint;
begin
       perform another_func(my_arg); -- same as perform my_schema.another_func(my_arg);
       select id into v_id
       from kernel.users -- table name is qualified with kernel schema name
       where login = my_arg;
       -- the rest is skipped...
end
$$ language plpgsql set search_path to my_schema;
```

Overriding the default function permissions

By default, PostgreSQL functions use the current database user's permissions, like Oracle's "AUTHID CURRENT_USER" option (Oracle default is "AUTHID DEFINER"). To emulate the Oracle behavior, a function should override the security option as follows:

```
create or replace function my_secure_func() returns void as $$
begin
  -- call here any functions available to the superuser
end
$$ language plpgsql security definer; -- default is security invoker
```

Emulating the deferred check constraints

Consider a simple check constraint such as **quantity >= reserve_quantity** on **ultima.vtb_stock** table. The immediate check constraint is supported out of the box, but the deferred version take a special kind of trigger to emulate:

```
-- create the trigger function to check the specified condition
create or replace function ultima.vtb_stock_resqty_chk_trigger() returns trigger as $$
begin
	if not new.quantity >= new.reserve_quantity then
		raise exception 'Deferred constraint violation.
			Table name: ultima.vtb_stock
			Constraint: vtb_stock_resqty_chk
			Condition: quantity >= reserve_quantity
			Data row: %', new
		using errcode = '23514'; -- check constraint violation error code
	end if;
	return new;
end;
$$ language plpgsql;

-- create deferred constraint trigger instead of the check constraint
drop trigger if exists vtb_stock_resqty_chk_trigger on ultima.vtb_stock;
create constraint trigger vtb_stock_resqty_chk_trigger
	after insert or update on ultima.vtb_stock
	deferrable initially deferred
	for each row
	execute procedure ultima.vtb_stock_resqty_chk_trigger();
```

Emulating Oracle-style global temporary tables

PostgreSQL semantic of temporary tables is substantially different from that of Oracle. Here is a brief summary:

- Oracle temporary tables are permanent, so their structure is static and visible to all users, and the content is temporary.
- In PostgreSQL, temporary table is created before each use. Both the structure and the content of a temp table is local for a database backend (a process) which created the table. PostgreSQL temporary tables are dropped either at the end of a session or at the end of a transaction.
- Oracle temporary tables are always defined within a user-specified schema.
- PostgreSQL temporary tables cannot be defined within user's schema, they always use a special (implicit) temporary schema instead.

Pack_temp schema contains the emulation library for the Oracle-style temporary tables. There are two functions:

- create_permanent_temp_table(table_name [, schema_name]);
- drop_permanent_temp_table(table_name [, schema_name]);

Creating a permanent temporary table is done in two steps:

1. Create a normal PostgreSQL native temporary table (that will be deleted at the end of transaction).
2. Use create_permanent_temp_table function to convert the temporary table into permanent one:

```
create temporary table if not exists another_temp_table
(
```

```
    first_name varchar,
    last_name varchar,
    date timestamp(0) with time zone,
    primary key(first_name, last_name)
)
on commit drop;

-- create my_schema.another_temp_table
select pack_temp.create_permanent_temp_table('another_temp_table', 'my_schema');

-- or create another_temp_table in the current schema
-- select create_permanent_temp_table('another_temp_table');

-- don't forget to commit: PostgreSQL DDL is transactional
commit;
```

The created object is a view that closely emulates the behavior of the Oracle-style global temporary table. To drop it, use drop_permanent_temp_table function.

Multiple active DataReaders sharing a database connection

This is the most annoying restriction of PostgreSQL: each connection allows only one DataReader to be opened at a time. New query cannot be executed until the last reader is closed. It seems to be the limitation of the network protocol, so any database driver for PostgreSQL has this restriction in place. The issue keeps popping up in the application services, LINQ queries and SQL queries in many different forms. Here are but a few common cases:

1. LINQ-query references a constant (or a service call). The query opens the first DataReader, the service tries to open the second and fails with an exception. To overcome the limitation, read the constant into a local variable before issuing the query (or call the service after fetching the query results). Example:

```
// before
var query =
       from a in DataContext.GetTable<Agent>()
       where a.ID = Constants.TestAgentID
       select a;

// after
var testAgentId = Constants.TestAgentID;
var query =
       from a in DataContext.GetTable<Agent>()
       where a.ID = testAgentId
       select a;
```

2. LINQ query results are processed in a loop, but the loop body executes another query, either LINQ or SQL. How to overcome: materialize the results of the query into an array or a list, and iterate over the results when the query is completed. Example:

```
// before
foreach (var langId in DataContext.GetTable<Language>().Select(x => x.ID))
{
       using (LanguageService.UseLanguage(langId))
       {
              // do something language-specific
       }
}

// after
foreach (var langId in DataContext.GetTable<Language>().Select(x => x.ID).ToIDList())
{
```

```
        using (LanguageService.UseLanguage(langId))
        {
                // do something language-specific
        }
}
```

3. Using ToArray/ToList/ToIDList inside the LINQ query. To fix the issue, the query should be broken into parts:

```
// before
var dictionary = DataContext.GetTable<CalendarDayStatus>()
        .Where(d => dates.Contains(d.DT))
        .GroupBy(g => g.DT, e => e.StatusID)
        .ToDictionary(k => k.Key, e => e.ToIDList());

// after
var dictionary = DataContext.GetTable<CalendarDayStatus>()
        .Where(d => dates.Contains(d.DT))
        .GroupBy(g => g.DT, e => e.StatusID)
        .ToDictionary(p => p.Key);

var dict = dictionary.ToDictionary(p => p.Key, p => p.Value.ToIDList());
```

Unfortunately, these kind of errors are hard to detect statically. Each non-trivial LINQ query has to be thorougly tested to make sure it doesn't open multiple parallel DataReaders at once.