



# Dynamic XML Query Table Editor

Sebastian Skritek  
Wolfgang Aigner  
Silvia Miksch

**Vienna University of Technology**  
Institute of Software Technology & Interactive Systems (ISIS)

Authors: **Sebastian Skritek**  
**Wolfgang Aigner**  
**Silvia Miksch**

sebastian.skritekgmx.at, {aigner, silvia}@ifs.tuwien.ac.at  
<http://ieg.ifs.tuwien.ac.at>

Contact: **Vienna University of Technology**  
Institute of Software Technology & Interactive Systems (ISIS)  
Favoritenstraße 9-11/188  
A-1040 Vienna  
Austria, Europe  
Telephone: +43 1 58801 18833  
Telefax: +43 1 58801 18899  
Web <http://ieg.ifs.tuwien.ac.at>

# Dynamic XML Query Table Editor

Sebastian Skritek, 0226286  
Sebastian.Skritek@gmx.at

June 2006

10.0PR Projektpraktikum (mit Bakkalaureatsarbeit)  
a. o. Univ. Prof. Dr. Silvia Miksch  
Dipl.-Ing. Dr. Wolfgang Aigner

## **Abstract**

The “Dynamic XML Query Table Editor” (DQT) is a web application for the management and presentation of data collections. It was implemented within a practical course using the “Ajax” approach, a new way to create web applications that perform like desktop applications. This thesis contains the documentation for DQT: A description of the DQT features, an installation guide, an explanation of the configuration possibilities, a user manual and a high level documentation of the DQT implementation.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Background and Motivations</b>	<b>5</b>
<b>3</b>	<b>Requirements and Design Goals</b>	<b>6</b>
<b>4</b>	<b>Application Areas and Target Users</b>	<b>6</b>
<b>5</b>	<b>DQT - Features</b>	<b>7</b>
<b>6</b>	<b>Limitations of DQT</b>	<b>7</b>
<b>7</b>	<b>Basic Structure</b>	<b>8</b>
<b>8</b>	<b>Installation</b>	<b>10</b>
8.1	System Requirements . . . . .	10
8.2	Directory Structure of the complete DQT Package . . . . .	10
8.3	Installation Steps . . . . .	11
<b>9</b>	<b>Configuration of the Dynamic XML Query Table Editor</b>	<b>14</b>
9.1	Configuration Step by Step . . . . .	14
9.2	Defining the Table Structure . . . . .	14
9.2.1	”Table Model” . . . . .	14
9.2.2	Table Structure Definition File . . . . .	16
9.3	Configuring the Database . . . . .	20
9.3.1	Mapping of the Table Structure XML File to the Database . . . . .	20
9.4	Defining the Attributes to be Displayed . . . . .	20
9.5	Adding/Removing an Attribute . . . . .	21
9.6	Data Types . . . . .	21
9.6.1	Basic Types . . . . .	22
9.6.2	Extended Types . . . . .	25
9.7	Export . . . . .	29
9.7.1	Export Format Syntax . . . . .	29
9.7.2	Export Format Semantic . . . . .	31
9.7.3	Export Stylesheets . . . . .	33
9.8	Configuration File <applRoot>/conf/config.php . . . . .	34
9.9	Managing Different Data Collections . . . . .	34
9.9.1	Handling of Different Data Collections within one DQT Installation	34
9.9.2	Adding a new Data Collection . . . . .	36
9.10	User, User Rights and User Related Configurations . . . . .	37
9.10.1	User Rights . . . . .	37
9.10.2	Default (guest) . . . . .	39
9.10.3	Management and Storage of User Data . . . . .	40
9.10.4	Navigation Bar . . . . .	41
9.11	index.php . . . . .	42



<b>10 User Manual</b>	<b>43</b>
10.1 Starting DQT . . . . .	43
10.2 Main Window . . . . .	43
10.2.1 Navigation Bar . . . . .	43
10.2.2 Content Area . . . . .	45
10.3 Edit Page . . . . .	45
10.3.1 Edit Tables . . . . .	46
10.3.2 Add New Entries . . . . .	48
10.3.3 Highlight an Entry . . . . .	48
10.3.4 Save Changes . . . . .	49
10.3.5 Undo Unsaved Changes . . . . .	50
10.3.6 Delete Entries . . . . .	50
10.3.7 Different Data Type User Interfaces . . . . .	50
10.4 View Page . . . . .	55
10.4.1 View Area . . . . .	56
10.4.2 Sorting the Table . . . . .	59
10.4.3 Hide/Show Rows . . . . .	59
10.4.4 Hide/Show Columns . . . . .	60
10.4.5 Search Area . . . . .	61
10.4.6 Export Area . . . . .	64
10.4.7 Export . . . . .	65
10.5 Detail-/Entry Window . . . . .	65
<b>11 Documentation of the Implementation</b>	<b>67</b>
11.1 Application Structure . . . . .	67
11.1.1 Components of DQT . . . . .	67
11.1.2 Overview over the Configuration Files . . . . .	67
11.1.3 Server . . . . .	68
11.1.4 Client . . . . .	80
11.2 Sequence diagrams and used data formats . . . . .	97
11.2.1 Load index page . . . . .	97
11.2.2 Login . . . . .	98
11.2.3 Logout . . . . .	100
11.2.4 Load the Edit Page . . . . .	102
11.2.5 Load Edit Table . . . . .	103
11.2.6 Load Edit Data . . . . .	106
11.2.7 Load Screenshot . . . . .	108
11.2.8 Save . . . . .	109
11.2.9 Add Screenshot . . . . .	111
11.2.10 Delete . . . . .	112
11.2.11 Delete Screenshot . . . . .	114
11.2.12 Load the View Page . . . . .	115
11.2.13 Load View Table . . . . .	116
11.2.14 Load View Data . . . . .	120
11.2.15 Search . . . . .	121
11.2.16 Export . . . . .	124
11.2.17 Screenshots . . . . .	126

11.3 To-do's and Known Bugs . . . . .	128
11.3.1 Known Bugs . . . . .	128
11.3.2 To-do's and Further Extensions . . . . .	129
11.4 Documentation of Design-Decisions . . . . .	130
<b>12 Conclusion</b>	<b>131</b>
<b>13 Appendix</b>	<b>133</b>
13.1 Database Structure of "Techniques for Visualization of Temporal Data" . .	133
13.1.1 EER Diagram: . . . . .	133
13.1.2 Database Description: . . . . .	134

# 1 Introduction

The “Dynamic XML Query Table Editor” (DQT) is a web/database application that was developed using the new “Ajax” [Garret,2005a] approach. DQT can be used for the management of data collections as well as for accessing (viewing, searching, exporting) the stored information. As DQT is a web application, it can also be used for the presentation of such data bases on the web.

Using the “Ajax” approach provides a simplified handling of DQT for the user compared to “traditional” web applications.

In those “traditional” web application, sending and receiving of data is done by sending a request to a webserver. In response, the webserver returns a new complete webpage, that replaces the existing one. The “Ajax” approach enables DQT to send data to/load data from the server without this need to load a new webpage. Therefore, although it runs without any plugins (only JavaScript is needed) in the webbrowser, DQT behaves like a desktop application. The user can save, delete or load data to/from the server without any interruptions due to page reloads.

The functionality of DQT includes editing (insert, update, delete) and presentation of stored data. It also provides a search functionality in the data sets and a data export. The exported data is XML formatted, but this output can be transformed by DQT using user defined XSLT stylesheets. A simple user management system allows to control the access to these functionalities.

This thesis contains the high level documentation of DQT.

In the first sections (2-7) of this documentation, background information about DQT is given. This includes the motivation for the developement of DQT (section 2), the design goals (section 3), the target users and application areas (section 4), the features (section 5) and limitations (section 6) of DQT as well as a short description of its basic structure (section 7).

Necessary requirements and steps for installing DQT on a webserver are described in section 8.

Section 9 describes the individual configuration of a DQT installation and explains specific implementation details as well as background information for configuration.

Unlike the other sections, the user manual in section 10 is intended for DQT end-users and describes the handling of the client side of DQT.

Section 11 contains information about the implementation of DQT for experienced users who want to change or extend DQT.

In the appendix, an overview of the tables used for saving the specific data collected by Wolfgang Aigner [Aigner,2006] is given.

## 2 Background and Motivations

In 2005, Wolfgang Aigner was working on his PhD Thesis about visualization techniques for temporal data [Aigner,2006]. Within that thesis, different visualization techniques have been classified by a set of approximately 40 properties. These information were stored in tables like the one shown in Table 1, formatted and saved as LaTeX files.

The rows of the table represent information about the different techniques, while the columns contain the attributes by which the techniques are classified.

Handling of these tables and the data stored in them was quite unhandy; data-sharing

	attribute 1		attribute 2		attribute 3		attribute 4		attribute 5		attribute 6		attribute 7		attribute 8	
	value	value	value	value	value	value	value	value	value	value	value	value	value	value	value	value
entry(Method 1)	•		•		•		•	•			•		•	•		
entry(Method 2)	•		•		•		•	•			•		•			
entry(Method 3)	•	•	•		•		•				•		•	•		
entry(Method 4)	•		•		•			◦			•		◦	•	•	

• ...completely fulfilled, ◦ ...partly or implicitly fulfilled

Table 1: Example of the existing LaTeX-tables

was complicated and search for specific entries was totally impossible. Therefore, a new application was strongly required that allows a more comfortable handling. Such an application, the “Dynamic XML Query Table Editor” (DQT), has been developed within a 12.5 ETCS practical course and is described within this bachelor thesis.

### 3 Requirements and Design Goals

The application for collected data management ought to provide following features:

- easy editing of the stored data (including inserting and deleting of data)
- data access via the www (for presentation as well as for management) using available web-browsers
- a search functionality to retrieve only entries that fulfill certain criteria
- a possibility to (easily) export the data into other formats
- a simple user and rights management system to allow restricted access to certain features of the application

Additional design goals were an easy configuration (adding or removing attributes/classification properties), and the possibility to adopt the application to similar tasks with moderate effort.

### 4 Application Areas and Target Users

The DQT can be used for management of any data that can be well presented tabularly within one table. This means the data consists of attributes (the columns) and entities

(the rows), that are classified or described by those attributes.

DQT is mainly intended to manage 1:1 relations between the stored entities and their attributes. This means that for each entry in the table (row), exactly one value for each attribute (column) exists. DQT also supports some simple list types, that handle short lists as one value.

Typical applications for DQT are classification tasks (like the one DQT has been originally created for) or comparisons of entities with respect to certain criteria (e.g., product comparisons). So DQT can assist creation, management and publication of databases containing collected data of a specialized domain.

Target users (as hoster of the DQT) are primary persons that already have or want to create such a data collection. Originally, target users were supposed to be experienced IT professionals. Therefore, for hosting and configuring DQT, skills in XML, SQL and XSLT are needed.

But also for users accessing a hosted version of DQT, some pre-knowledge is assumed. This is because the DQT client (a JavaScript application running in standard web-browsers) requires JavaScript to be enabled, what is not fulfilled for all users. But the assumption was made that the target users are able to create an environment (or have access to) that is capable of running the DQT client.

The DQT client is only tested with the two mostly applied web-browsers, Firefox (1.5) and IE 6 (having some minor problems in IE6), but might run in other browsers too (especially in all Mozilla/Firefox based browsers).

## 5 DQT - Features

DQT is a web application that provides an easy way of managing (insert, edit, delete) table data saved in a database. The use of the "Ajax" [Garret,2005a] approach allows client-server communication to be performed in the "background". Because of this, loading or saving data neither interrupts the user's online workflow nor requires a reload of the current webpage. The behaviour of DQT is therefore similar to the behaviour of desktop applications, without needing any browser plugins. DQT also contains a module for tabulating the stored data. The use of JavaScript also allows dynamically display of certain selected rows and columns of the table. It therefore provides comparison of data subsets that are currently of interest. Also, the displayed data can be exported to an XML formatted output which can be further transformed using XSLT to any output format needed. A search function allows to display entries that fulfill the defined constraints. Again, search and export requests are performed in the background, without interrupting the user's workflow.

To restrict the access to certain features, a simple user- and rights management system was implemented. It allows to grant or deny user access to the different features of DQT. For displaying each entry in detail and separated from others, DQT offers a page that only contains the information of one data row.

## 6 Limitations of DQT

This section describes the "not"- features of DQT, and the current limitations of the software.

- DQT is not a management tool for the structure of the used database (like phpMyAdmin [PHPMyAdmin,2006] for example). This must be done otherwise, as DQT assumes a correctly configured database.
- DQT should only be used to manage small/medium sized data bases (with a typical upper limit of approximately 7000 - 10 000 table cells). This number varies depending on the clients used, as rendering of the tables by the browser requires considerable time and the presentation of such tables needs a lot of memory. It is therefore no management tool for databases simply collecting big amounts of data for later evaluation.
- DQT does not support statistical evaluation of the stored data.
- There is no concurrency control for the editing of entries. If two users change data at the same time, they do not know about each other and are not informed about the changes the other one made.
- No graphical interface exists for installation or configuration of DQT. Skills in XML, SQL and probably XSLT are needed to configure DQT.
- The export feature allows to transform the output using XSLT to different data formats, but no XSLT stylesheet is included in the DQT. They therefore first need to be created.

## 7 Basic Structure

As a web application, DQT consists both of a server side, and a client software. The latter is loaded by a web browser (Firefox, Internet Explorer 6). The application logic on the server is implemented using php and intentionally runs on a webserver. The data is stored using a MySQL database. (See Fig. 1. A detailed description of this figure is given in section 11.1.1 “Components of DQT”)

Before starting the implementation of DQT, the author of this thesis has analyzed within a term paper whether the possible use of Ajax [Garret,2005a] for the implementation of the client part, or whether a “traditional” website interface should be implemented. The main advantage of Ajax is, that it enables the user interface to adopt user actions without reloading the entire page, even if this action requires data to be sent/retrieved to/from the server.

The conclusion was, that using Ajax would allow an improved user interface and would result in a better user acceptance than a traditional web interface. Therefore use Ajax was chosen for the development of DQT [Skritek,2005].

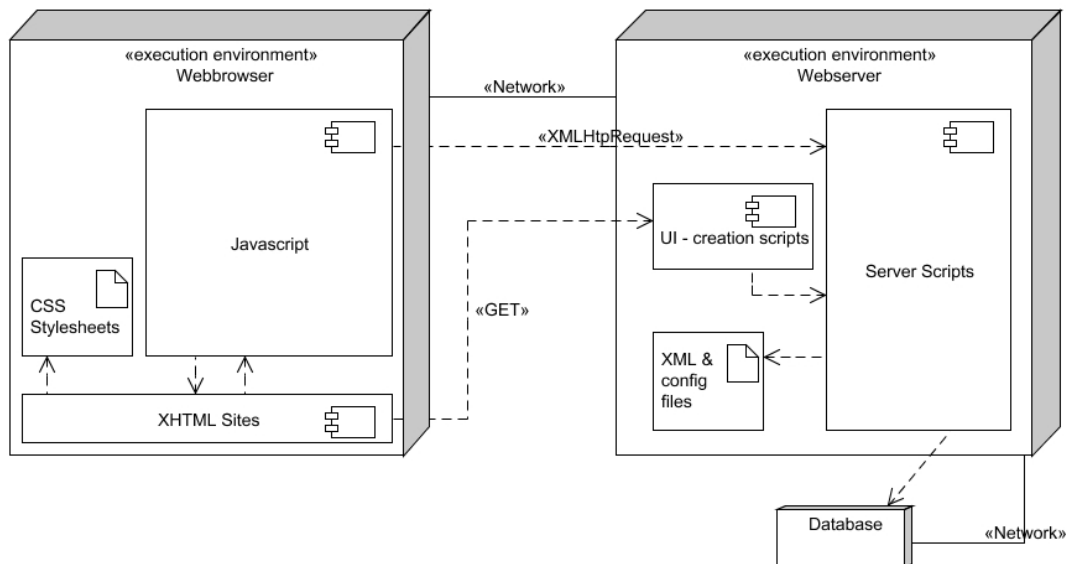


Figure 1: The basic components of DQT

## 8 Installation

This section describes the system requirements and steps necessary to install DQT.

### 8.1 System Requirements

#### Server:

On the server side, following software is necessary to run DQT:

- MySQL 5.0 (might run with older versions, too)
- PHP 5.1 (will not run with older versions) with following modules installed and active:
  - mysql
  - xsl (XSL (libxslt, tested with version 1.1.15) and EXSLT (libexslt, tested with version 0.8.12))
- a webserver (tested with apache httpd 2.0.54)

#### Client:

The client needs following software to run the application:

- Browser: Firefox (tested with 1.5; may run with 1.0, too) or Internet Explorer (tested with 6.0)
- Not tested with other browsers; minimal browser requirements:
  - support of JavaScript
  - support of the XMLHttpRequest
  - support of DHTML

Hint: The configuration of XAMPP [XAMPP,2006] (at least version 1.5.1 for windows) contains all modules necessary.

### 8.2 Directory Structure of the complete DQT Package

<i>/src</i>	contains the full-length sourcecode files
<i>/lib</i>	contains files from external libraries
<i>/bin</i>	contains the executables and used libraries (this directory is referred to as <i>&lt;applRoot&gt;</i> (application Root) in this paper)
<i>/doc</i>	documentation (javadoc and further documentation)
<i>/data</i>	(example) datafiles
<i>/html</i>	webpage (brief project information)

The */src* and the */bin* directories contain the same files and structures, but the code in the */bin* directory is compressed ( all comments and unnecessary whitespaces are removed).



The content of both directories is:

<i>/src</i>	all php files needed for the server part of DQT
<i>/src/conf</i>	configuration files for DQT
<i>/src/cssStyles</i>	CSS files used
<i>/src/helpfiles</i>	XHTML files for the online help
<i>/src/icons</i>	all icons used in the application; png format
<i>/src/install</i>	SQL files with the initial database content and pre-defined configuration files
<i>/src/js</i>	all javascript files
<i>/src/logs</i>	the directory where DQT stores its logfiles
<i>/src/screens</i>	the directory where DQT stores the screenshot - images
<i>/src/XMLData</i>	XSLT stylesheets to transform the structure definition of the table and the structure definition itself
<i>/src/XMLData/exportStylesheets</i>	XSLT stylesheets for export of data into different formats

## 8.3 Installation Steps

1. Unpack (unzip/unrar/...) the */bin/DynamicXMLQueryTableEditor-<version>.(zip—rar)* archive into the destination folder (*<applRoot>*).
  - Make sure the webserver has write permission on the *<applRoot>/logs* directory (as the logfiles of the server side of DQT are stored there)
  - The webserver also needs write permissions on the *<applRoot>/screens* directory (as the screenshots are stored there)
2. Create a new database on MySQL for the application (optional)
3. Go to *<applRoot>/conf/* and make the following changes on:

### 3.1 *config.php*

- line 11: change `$conf_urlPrefix` to the base-URL of your installation of DQT (must end with `/` )
- lines 16-19: update the values corresponding to your database settings:
  - `$db_name`: contains the name of the database
  - `$db_user`: contains the username that shall be used to connect to the database
  - `$db_pw`: the password for the defined user
  - `$db_host`: The host name or address where the database is placed.
- line 29: `$conf_appName` defines the content of the `<TITLE>` tag of the webpages.

### 3.2 *config.xml*

- line 5: change the value of `<urlPrefix>` to the same value as `$conf_urlPrefix`
4. Run the *<applRoot>/install/initialize.sql* script on the database. This will create a user “admin” with full rights (pw = “admin”). A test database table is created too.

- Open your webbrowser (best choice for use of DQT is Firefox), and try to open DQT. The screen should look like as shown in Figure 2.

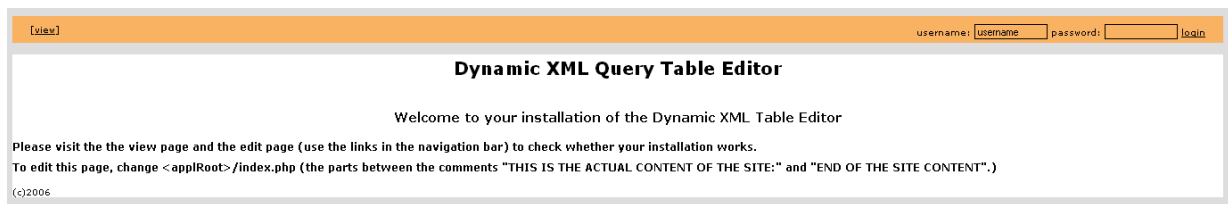


Figure 2: Screenshot of the index page like after successful DQT-installation

- Login as admin user (pw = “admin”), and go to the “view” page. This should look like as shown in Figure 3.

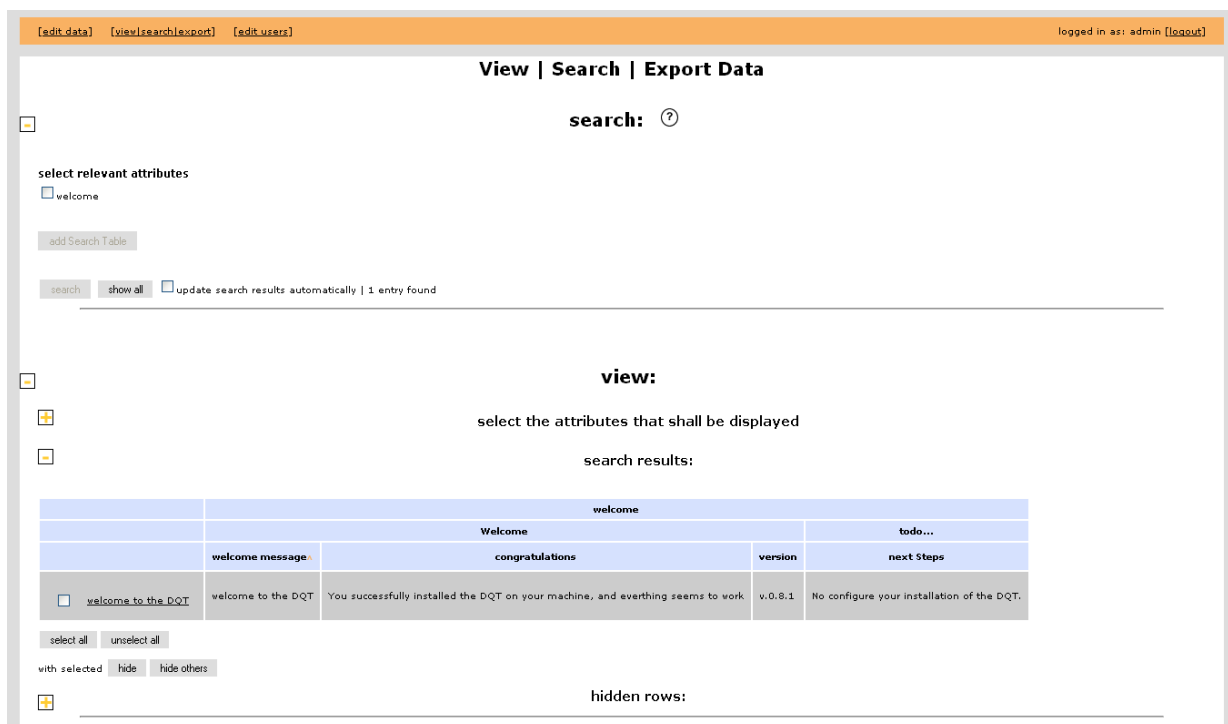


Figure 3: Screenshot of the view page after successful installation

- Go to the “edit” page. It should look similar to Figure 4.

If this is the fact: Congratulation. You have just finished installing the DQT.

## Setting up a Data Collection

Next, one has to configure its installation. To use the collected data about “Techniques for Visualization of temporal data” (by Wolfgang Aigner), follow the next steps. If you want to set up your own data structure, please go to the “Configuration of DQT” section of this paper.

Hint: If you made an error in setting the database access data (e.g. given a wrong password), php can be very obstinate in using these erroneous values.

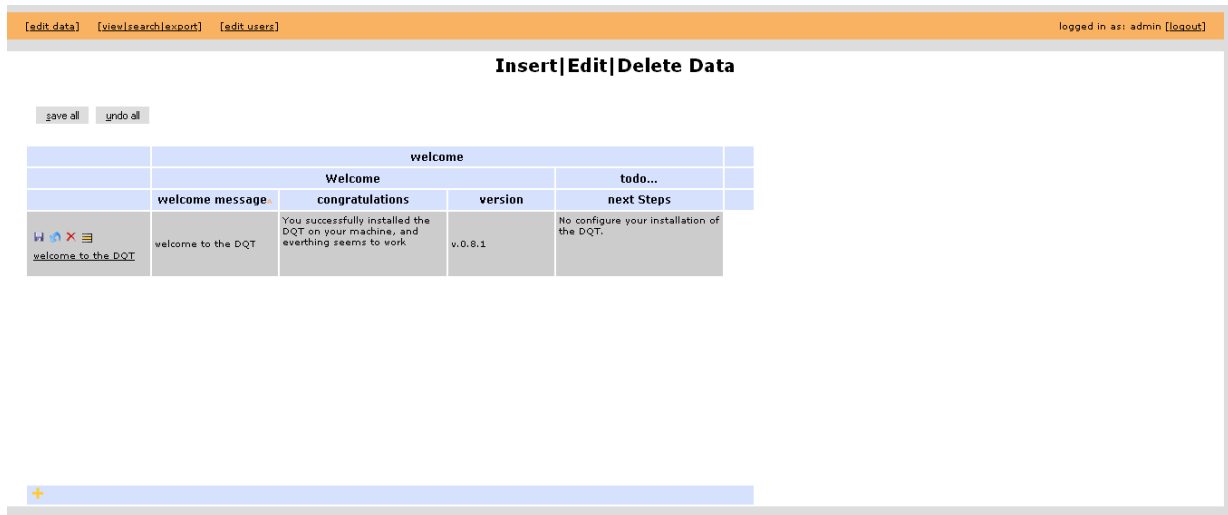


Figure 4: Screenshot of the edit page after successful installation

Goto `<applRoot>/sessionManagement.php` and uncomment line 34. Reload the application (by pressing the reload function of the browser), and then comment the line out again. This should force php to use the new values from `<applRoot>/conf/config.php`

## Installing the Data Collection “Techniques for Visualization of Temporal Data”

1. DQT must have been successfully installed
2. Run the script `<applRoot>/install/vistechs/initializeVisTechs.sql` on the database. This will remove the test database and insert information about 84 visualization techniques to the database
3. Copy the file `<applRoot>/install/vistechs/conf/config.php` to the folder `<applRoot>/conf` (overwrite the existing `config.php` file). Do step 3 of section “Installation Steps” again
4. Copy all files from `<applRoot>/install/vistechs/XMLData` to `<applRoot>/XMLData` (overwrite the existing files)
5. Copy all files from `<applRoot>/install/vistechs/XMLData/exportStylesheets` to `<applRoot>/XMLData/exportStylesheets`
6. (Re-)Start the DQT client (This will reload the server as well)
7. If DQT now contains the new data: Congratulations. You have just completed the configuration of the DQT with the data about “Techniques for Visualization of Temporal data”

## 9 Configuration of the Dynamic XML Query Table Editor

This section explains the configuration features of DQT. First an overview over the configuration steps is given. Then all steps and actions are explained in detail.

### 9.1 Configuration Step by Step

1. Installation of DQT (see previous section 8: “Installation”)
2. Defining the structure of the data base (see section 9.2: “Defining the Table Structure”)
3. Creating the database tables (see section 9.3: “Configuring the Database”)
4. Defining filters for which attributes shall be shown on certain pages (see section 9.4: “Defining Attributes to be Displayed”)
5. Updating the file `<applRoot>/conf/config.php` file (see section 9.8: “Configuration File `<applRoot>/conf/config.php`”)
6. Creating and adding export stylesheets (see section 9.7: “Export”)
7. Further configurations

### 9.2 Defining the Table Structure

This section describes how the table structures used to represent the stored data can be defined. Therefore, first an explanation of the table model (including the exact meaning of “table structure” and of which parts this structure consists) is given. As the table structure is defined by an XML formatted file, afterwards a description is given how it is mapped to that file.

#### 9.2.1 ”Table Model”

As DQT is designed to manage data represented in table form, the main structure of DQT is the table. Similar to tables in relational database systems, DQT understands a table as a tuple of attributes (the table columns). Therefore, with respect to the description of a table, in this thesis “attribute” and “column” is used synonymously). Each row contains exactly one value for each table column. Unlike relational database systems, DQT allows to organize columns into groups, and also to combine groups to larger groups. This can be used to structure the attributes. For each group or column, a unique identifier and a name used for labeling the column/group in the table exists. Furthermore, for each column or group a text may exist that describes the column/group. The structure of the groups must result in a tree, so each column or group can be only part of exactly one parent group. An example for such a grouping of columns is given in Figure 5. Each black dot represents one group of columns/groups.

DQT provides different levels for grouping attributes/columns (Fig. 6):

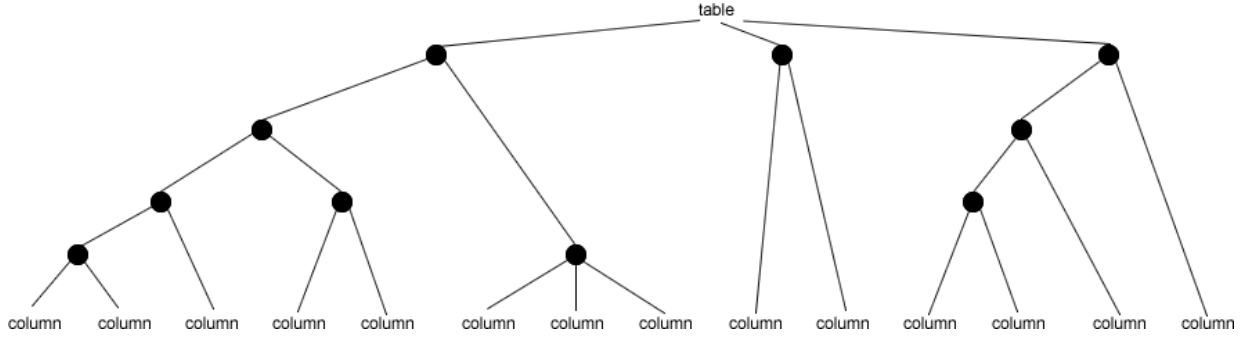


Figure 5: Example of column grouping for a table. Each black dot represents a group.

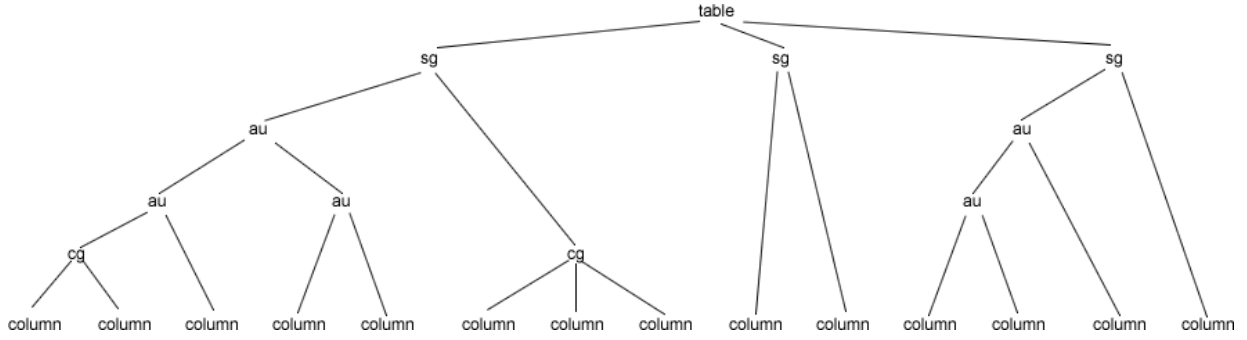


Figure 6: A possible DQT table structure for the grouping shown in Fig. 5 (cg... column group, au... attribute unit, sg... semantic group).

## Column Groups

At the lowest level, attributes can be combined to “column groups” (or “attribute groups”). A column group may contain one or more columns. If a column is not explicitly assigned to a column group, an “anonymous” column group that only contains this column is implicitly created by the application. The column groups can be used for example to put the name of a column into a context (e.g., in [Aigner,2006] there exist the attributes “linear”, “periodic” and “branching” that are combined in the column group “organization”). This may allow to use shorter names (e.g. “linear” instead of “linear organization”) for the columns, and to make the structure and coherences of the attributes more clearly.

## Semantic Groups

The highest level of grouping is called “semantic group”. Each column must be exactly part of one semantic group (not directly, but transitive over groups it belongs to). Semantic groups are intended to define the rough domain of the columns belonging to the group, and to distinguish between columns whose semantic fields are completely different. Semantic groups can be used to partition the set of attributes into independent subsets, each representing completely different features of the stored entities. Because of the great differences that may exist between the semantic groups, for each semantic group an own database table is created. The idea behind this was, that for a stored entry, very often not the values for all available attributes will be accessed. Instead, only subsets of attributes which are part of the same semantic groups are of interest. For example, some of the semantic groups in [Aigner,2006] are called “general information” (attributes belonging to this groups are “name”, “description”, “references”,...), “data aspects” or “supported tasks

and application areas”.

### Attribute Units

Between semantic groups and column groups, an arbitrary number of so called “attribute units” can be defined. These attribute units can be used to further restrict the domain of the attributes, in order to give a more detailed explanation of the classification or to distinguish more clearly between those attributes.

Unlike column groups or semantic groups, where each column is only allowed to belong to (a maximum of) one group, now each column can belong to an arbitrary number of nested attribute units. An attribute unit may therefore not only contain columns or column groups, but can also contain attribute units.

On the edit page of DQT, only semantic groups, column groups and columns are shown in the table headers (to restrict the headers size). On the view page, only semantic groups, columns groups and columns can be activated/deactivated directly by the user. This is again made to restrict the user interface complexity.

The grouping of the columns shown in Figure 5 could be realized in DQT as shown in Figure 6.

#### 9.2.2 Table Structure Definition File

The structure of the table attributes is defined in an XML file located in `<applRoot>/XMLData`. In the standard installation, it is named `tableStructure.xml`. Herein the columns, column groups, attribute units and semantic groups can be defined. “Anonymous” attribute units or column groups need not to be defined in this file. This defines a tree, with the columns as leaf nodes, typically similar to Fig. 6.

The defined grouping of the columns is visualized in the table head. Therefore, for each depth in the tree, one row is inserted to the table head. A row contains one cell for each node of the tree in the corresponding depth. The number of columns being descendants of the node denote the colspan of the cell.

As Fig. 6 shows, because the different leaf nodes of the tree have different depths, if this transformation is performed on a typical tree defined by the table structure XML file, not all rows would have the same number of cells.

Therefore, whenever the XML file is processed by the application, the tree is completed, so that a table can be built from it. For this “completion”, following transformation is applied:

- For all missing column groups “anonymous column groups” are inserted into the tree. Thereby “missing” means, that for a column no column group was defined.
- For all missing attribute units, “anonymous attribute units” are inserted into the tree. Thereby, for each column x, the number of “missing” attribute units equals the difference between the length of the longest path within the tree from the root to a leaf node and the path length from the root node to the leaf node x. To fill in the required number of “anonymous attribute units”, the user defined attribute units (“named attribute units”) are pushed toward the root of the tree, and then the “anonymous attribute units” are inserted between the user defined attribute units and the column groups.

For example, the tree shown in Fig. 6 is implicitly mapped to the tree shown in Fig. 7 by DQT. This completion of the tree is only done on a logically level, the definition in the XML-file remains unchanged.

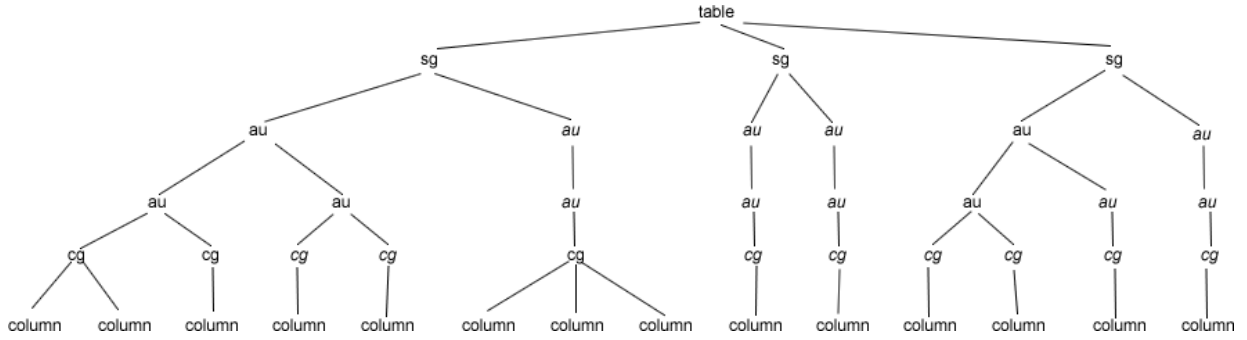


Figure 7: The “completed” table structure from the table shown in Fig. 6. (cg... column group, *cg*... anonym column group, au... attribute unit, *au*... anonym attribute unit, sg... semantic group)

The completed tree of Fig. 7 can then easily be transformed to the table head shown in Fig. 8.

sg - name					sg - name		sg - name			
au - name							au - name			
au - name		au - name					au - name			
cg - name				cg-name						
col - name	col - name	col - name	col - name	col - name	col - name	col - name	col - name	col - name	col - name	col - name

Figure 8: The completed tree of Fig. 7 is transformed to this table head

## Table Structure XML File Syntax

The definition of the table structure must be valid for the following DTD:

Listing 1: DTD for the table structure defintion file

```
<!ELEMENT attributes (semanticGroup+)>

<!ELEMENT semanticGroup (description | attribute | attributeUnit
| attributeGroup)+>
<!ATTLIST semanticGroup
  name CDATA #REQUIRED
  tName CDATA #REQUIRED
  maud CDATA #IMPLIED
>

<!ELEMENT attributeUnit (description | attribute | attributeUnit
| attributeGroup)+>
<!ATTLIST attributeUnit
  name CDATA #REQUIRED
  shortName CDATA #REQUIRED
```

```

>
<!ELEMENT attributeGroup (description | attribute)+>
<!ATTLIST attributeGroup
    name CDATA #REQUIRED
    shortName CDATA #REQUIRED
>

<!ELEMENT attribute (name, shortName, type, description?, value*)
>
<!ELEMENT value (name, shortName, order)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT order (#PCDATA)>
<!ELEMENT shortName (#PCDATA)>
<!ELEMENT type (fpn | int | string | stringlist | urllist | text
    | enum | set | md5 | extended)>

```

## Table Structure XML File Semantic

The information stored in the elements defined by the DTD given above is as follows:

### semantic group:

This tag defines a semantic group. As mentioned above, each column (attribute) must belong to exactly one semantic group.

The optional *description* tag may contain some textual description of the semantic group. If more than one description tag is defined for one semantic group, only the content of the first is displayed.

The “*tName*” (abbr. for tableName) attribute of the semanticGroup tag must contain the identifier of the semantic group. This identifier must be unique within all identifiers of semantic groups defined for this installation of DQT. Note, that it is therefor not enough for the identifier to be unique within the identifiers of the semantic groups defined in the same file. DQT support more than one table structure definition files for the DQT installation (see “Managing different data collections”). A semantic group identifier must therefore be unique within all these files. Further, the identifier must not start with “sys\_” (abbr. for system; this prefix is reserved for identifiers used by DQT) and must not contain the “;” (semi-colon) character.

The “*name*” attribute may contain any text. It stores a name that is used to label the semantic group for the user. It is, for example, shown in the table head, and should therefore be reasonable short.

The “*maud*” attribute must hold the deepest nesting of <attributeUnit> elements in this semantic group. (For example <semanticGroup> <attributeUnit/> </semanticGroup> => *maud* = 1; <semanticGroup> <attributeUnit> <attributeUnit/> </attributeUnit> </semanticGroup> => *maud* = 2; ...)

### attributeUnit:

This tag defines an attribute unit.

The “*shortName*” attribute contains the identifier for the attribute unit. This identifier must be unique for all attributeUnit elements within the same semantic group. The shortNames of the attributeUnits again must neither start with “sys\_” nor contain the “;”



character.

To each “anonymous attribute unit” DQT applies an identifier of the form {shortName of the attribute}\_aau{depth}, where...

- {shortName of the attribute} is the identifier of the attribute that belongs to this attribute unit.
- “aau” is short for “anonymous attribute unit”.
- {depth} indicates at which depth in the tree the anonymous unit is added. (Counting starts at 0.)

To avoid name clashes, the postfix “\_aau{number}” should not be used for named attribute units as well. Otherwise the user is responsible to assure that no name clash occurs.

The “*name*” attribute defines the text that is displayed to label the attribute unit for the user. Again it should be reasonable short, as this text is displayed in the table head.

As for the semantic group, the description element is optional. Only the content of the first description element is taken into account.

#### attributeGroup:

With this tag, an attribute (column-) group can be defined.

The “*shortName*” attribute defines an identifier for the attribute group. This identifier must be unique for all attribute groups within the semantic group. It must not start with “sys\_” and must not contain the “;” character. DQT creates for those attributes, for which no attribute group is defined, an anonymous attribute group. For these attribute groups, the identifier is created by appending “\_aag” (abbr. for anonym attribute group) to the attribute identifier. The identifier of a user defined attribute group should therefore not end with “\_aag”.

The “*name*” attribute may contain any string. This string is used to label the attribute group for the user. As it is displayed within all table heads where the attribute group is part of, it should contain only a few characters.

As for the semantic group, the *description* element is optional, and only the content of the first description element is taken into account.

#### attribute:

This element defines an attribute (one table column).

The content of the child element “*shortName*” defines the identifier for the attribute. Again it must be unique for all attributes within one semantic group.

The content of the “*name*” child element contains the string that is used to label the attribute for the user. To fit into the cell in the table head, it should be as short as possible.

The child-node “*type*” defines the data type of the column (see section “Data Types”).

The optional *description* element may contain arbitrary text. This text is used to provide the end-user of DQT further information about the attribute.

The “*value*” child element is only needed if the datatype of the attribute is “enum” or “set”. (See the description of these data types - section “Data Types” - for more information about this element.)

The first attribute of each semantic group must have the shortName and name “id”. This attribute must uniquely define each data row. (See next section 9.3 “Configuring the Database” for details.)

Each table structure XML file defines one table. (More than one table structure file may exist. See 9.9 “Managing Different Data Collections”.)

## 9.3 Configuring the Database

A data row (or table row) is a tuple that contains a value for each defined attribute in the XML file (= column of the table). These data rows are saved in the database.

### 9.3.1 Mapping of the Table Structure XML File to the Database

The table structure definition file(s) contain(s) all information which DQT needs to create the tables for the display and the export output. It also stores all information DQT needs to load entries from the database (or to save it to the database).

Therefore, the attribute (structure) needs to be mapped to the database tables. This is done according to the following rules:

Each semantic group is mapped to an own database table. The value of the `tName` attribute (the identifier) of the `semanticGroup` element is used as table name. Because all tables of one DQT installation DQT are stored in the same database, it is necessary that the value of the `tName` attribute is unique for all semantic groups of the application.

All attributes belonging to a semantic group and having one of the basic data types (see “Data Types” for details) are mapped to fields in the corresponding database table. (How attributes with an extended type are handled: see section “Extended Types”) Thereby, the value of the `<shortName>` element is used as field name. Thus, values of one table defined in the table structure XML file are saved in different database tables. So for each data row in the table, there must exist an entry in each of the corresponding database tables. To rebuild the complete data entry again, information is needed which rows in the different database tables belong together (a foreign key). Therefore, each database table must have a field named “id”. It must be the first field of each database table. This field is the mapping of the “id” attribute required in each semantic group (see “Table Structure XML File Semantic”). It must contain a unique value for each entry. This value is automatically assigned by DQT when a new entry is saved for the first time. It must not be defined by the user.

As mentioned before, this field must be the first in each database table. The order of the other fields is arbitrary.

Attributes of extended types are stored differently. Therefore for attributes with extended types, no fields in the database table needs to exist.

The attribute units and attribute groups are not mapped to the database.

## 9.4 Defining the Attributes to be Displayed

The tables shown on the “edit-” and “view page” are created using the information of the table structure XML file. This file contains the complete list of fields that are stored in the database. But maybe not all fields shall be displayed in all tables. For example, it would not make sense showing the “id” field in the “edit table”. (As the id is assigned by the system.)

Therefore, before the “view” or “edit” tables are created, the table structure XML file is transformed using an XSLT stylesheet. Using this stylesheets, it is possible to define filters that prevent copying of certain attributes to the resulting table. For both, the view and the edit table different stylesheets can be defined. The default stylesheets are stored in `<applRoot>/XMLData/TS2editTS.xsl` (for the edit tables) and `<applRoot>/XMLData/TS2viewTS.xsl` (for the view table).

The XSLT stylesheet used for the transformation can be defined in the `<applRoot>/conf/config.php` by the values `$conf_ts2editTs` and `$conf_ts2viewTs`. They must contain the name and location of the `*.xsl` file to use relative to `<applRoot>/XMLData/`. For a description of the structure of these filters, please see the inline documentation of the default files.

## 9.5 Adding/Removing an Attribute

Following, a step by step description for adding or removing of an attribute (with a basic data type) is given:

1. Change the database: Delete or add a field from/to the database table corresponding to the semantic group to which the new attribute belongs to.
2. Change XML file defining the table structure: Delete or add an `<attribute>` tag to the file. This tag must be a descendant of the `<semanticGroup>` tag the new attribute shall belong to. It must be the semantic group whose corresponding table has been updated in step 1. The child element `<shortName>` of the new `<attribute>` element must contain the name of the field added/removed from the database table. (See 9.2 “Defining the Table Structure” and 9.3 “Configuring the Database”.)
  - If a new attribute shall not be shown in the “edit” or the “view” table, update the filters in the TS2...TS.xsl files. (See 9.4 “Defining the Attributes to be Displayed”.)
3. Reload the application. If the application has been loaded before, php can be very obstinate in keeping the old table structure. Go to `<applRoot>/sessionManagement.php` and uncomment line 36. Then reload the application (using the reload function of the browser), comment this line out again, and reload the application once more. Now the new table structure should be completely loaded by the application.

To change the grouping of the attributes or to redefine the description texts, only the XML file storing the table structure definition needs to be changed. Afterwards DQT needs to be reloaded.

## 9.6 Data Types

To each attribute/column, a data type must be assigned. It restricts the values that can be entered into this column.

In DQT, two main datatype classes exist:

The

- “basic types” and the
- “extended types”.

The main difference between these data types is their implementation:

Basic types are a fixed part of the system, and are more deeply integrated into the software compared to extended types. Extended types shall provide an interface to append new datatypes to the system easily.

Another differentiator is the kind of data “basic” and “extended” types are able to store. The basic data types could be called scalar types, as their values can be saved in one database field. Also, for each basic type, only one value exists for each data entry. The screenshot type (which is an extended type) needs 5 database fields in an extra database table and a directory on the webserver to store the images in. There can be also more than one screenshot for each data entry.

This differentiator is only valid in the current version, but must not keep valid for data types implemented in the future. This classification of datatypes may help to decide whether a new datatype shall be implemented as extended or basic type.

On the client, each data type is implemented in an own object. Such an object is assigned to each table cell. This “type object” handles all actions from the cell object that are somehow related to or dependent on the datatype.

(All type objects are defined in the `<applRoot>/js/types.js` file)

### 9.6.1 Basic Types

In the current implementation, 9 basic types are provided. In this chapter, only details of the datatypes are explained which are important for the configuration of DQT (or for changing the implementation of DQT). Information how the datatypes are presented to the user and the user interfaces is explained in the user manual.

The data type of a column is defined in the table structure XML file. For each `<attribute>` element, the child element `<type>` must contain the name of the datatype.

#### **int (Object: IntType)**

The “int” type is capable of handling integer numbers. For the database, it is mapped to the SQL INTEGER (INT) type (at the MySQL database this allows values between -2147483647 and 2147483647). DQT allows to search for exact matches or to search for values within a given interval. For the comparison of two “int” values, the standard order of integer values is used.

#### **string (Object: StringType)**

The “string” type is used to store short string values (up to 200 characters). For the database, it is mapped to the VARCHAR type (with a maximum length of 200 characters). The string is allowed to contain any character defined by UTF-8. DQT allows fulltext search in attributes of this type (and therefore uses the fulltext search functionality of the MySQL database).

For the comparison of two “string” values, all characters are normalized to upper case and then an alphabetical comparison is performed (as it is provided by javascript).

#### **md5 (Object: StringType)**

The “md5” type is very similar to the “string” type. It stores md5 hashed strings [Rivest,1992]. Within the client, the md5 type is identical to the “string” type, as it uses the same implementation. On the server, whenever a md5 value is saved, the string that is received from the client is hashed before it is stored in the database.

In the database, the hashed string is stored as VARCHAR.

For md5 values, the same comparison algorithm is used as for “string” values.

**text (Object: TextType)**

The “text” type is intended to store longer string values. It is stored in the database as LONGTEXT value (MySQL database allows LONGTEXT values up to 4GB). The string may contain any character defined by UTF-8. DQT allows fulltext search in this data type (using fulltext search functionality of the MySQL database). For “text” values, the same comparison is used as for “string” values.

**stringlist (Object: StringlistType)**

The “stringlist” type is a primitive list type, whose entries can contain string values. The whole list is stored as a LONGTEXT field in the database. The different list entries are separated by a semi-colon (“;”). Because the list must be defined by the user in exactly this way (by typing the list entries separated by a “;” character to a text field), the list entries must not contain “;”.

DQT allows fulltext search within different list entries. That means it is possible to search for lists that contain at least one entry containing a certain string, and another entry containing another string.

When comparing two lists, first the length (that is the number of (not empty) list items) of the lists is compared. If two lists have the same length, an alphabetical comparison of the strings defining the whole list is used (after all characters are converted to upper case letters). If one list has more entries than the other, it is regarded to be bigger.

**urllist (Object: UrllistType)**

The “urllist” type is a variation of the “stringlist” type. The “urllist” type expects the list items to contain URLs. This does not lead to any difference how to edit the items, but when displaying the list, the list items are linked to the values defined in the items. Everything else is equal to the “stringlist” type.

**enum (Object: EnumType)**

The “enum” type allows to select one value out of a predefined set of values. This set can be defined by the host of the DQT.

This is done using the <value> tag that can be inserted as child node to the <attribute> value in the XML file, which defines the table structure.

For each chosen value, one <value> element must be defined. The child element <name> defines the string that is used to label the value towards the user. It can be any string value. The <shortName> element defines the identifier for the value. This is used to save the value in the database. It must be unique within the <value> element of the attribute, and must not contain the “,” character (as this leads to problems with the database). The <order> element should contain a number. This number is used for the comparison of two values. The value with the higher order number is regarded to be bigger than the value with the lower order number (  $v1 > v2 \iff \text{order}(v1) > \text{order}(v2)$  ).

This datatype is mapped to the ENUM type of the MySQL database. The set of defined values must be defined in the database as well. Additionally, a possibility to save an empty entry (“”) must be defined in the database.

For example, an attribute (with idName “att”) of type “enum” with the possible values “id1”, “id2” and “id3” is saved in a database field created by the SQL statement “att ENUM(‘,id1’,‘id2’,‘id3’)”. Whereby “id1”, ... are the values defined in the <shortName>

element of the <value> tag.

DQT allows to search for exact matches (that means for entries having a specific value of the set selected).

### **fpn (Object: FPNTYPE)**

The “fpn” type is a certain enum type. The attribute value of “fpn” types can be one out of three values: (completely) fulfilled (f), partly or implicitly fulfilled (p), not fulfilled (n). (So “fpn” is the acronym of these values.) These values are related to the attribute. They describe whether the property of the attribute is fulfilled by the entry.

Unlike the normal “enum” type, where the possible values can be defined by the host of DQT, these values are fixed. Also, “fpn” type offers a different user interface than the “enum” type. (See the user manual for details.)

For storing in the database, this type is mapped to an ENUM type with the possibilities 'f','p','n'. (ENUM('f','p','n')). These three letters are always used to describe/define the current value of an entry.

DQT allows to search for exact matches (entries with the value 'f', 'p' or 'n').

The comparison of two values of the “fpn”-type is done according to the following transitive order:

(completely) fulfilled > partly/implicitly fulfilled > not fulfilled.

### **set (Object: SetType)**

The “set” type allows to select an arbitrary number of values out of a predefined set of values. This set can be defined by the host of DQT in the table structure XML file .

For each value of the set, a <value> element must be appended to the corresponding <attribute> element. The <name> child element of the <value> element defines the string that is used to label the value towards the user. It may contain any string value.

The <shortName> element defines the identifier for the value. It is used to save the selected set of values in the database (and to identify the different values). This identifier needs to be unique within the values of an attribute, and must not contain the “,” character (as this leads to problems with the database).

The <order> element should contain a number that defines the order in which the values of the set shall be displayed on the screen. The values are shown in ascending order.

In the database, this type is mapped to the SET type. For each value defined in the table structure XML file, an entry in the set of possible values of the database must exist. The names of the set items defined in the database must equal the values of the <shortName> element defined in the table structure file.

DQT allows to search for entries, whose set of selected values is a superset of the set defined in the search request. So DQT returns all entries where at least the values defined by the search are selected.

When comparing the selected sets of two entries, first the number of selected entries is compared. The entry with more selected values is regarded to be bigger than the one with less entries selected. If both selected sets have the same size, for each set the identifiers of the selected values are combined by “,” and all letters are converted to uppercase letters. Then an alphabetical comparison between these two strings is made.

## Creating a new Basic Type

Following, a step by step description for creating/adding a new basic type, that can be saved in one table field, to DQT is given.

1. Define a new object that implements the new data type (type object). This object must handle all type-related actions of the cell object. Therefore, the following interface must be implemented:
  - `int compare(Cell myCell, Cell otherCell):`  
Takes two Cell objects as arguments; returns -1 if the value of the first cell is smaller than the value of the second cell, 0 if the values of the cells are equal, and 1 if the value of the first cell is bigger than the value of the second cell.
  - `void setValue(String value):`  
Stores the value passed in the “value” argument as current value of the corresponding cell object. The value is saved in the `actValue` field of the cell object. The correct representation of the value is displayed on the screen.

Furthermore, this type object must create and manage the user interface of the corresponding cell. That means the type object is responsible for the creation of the user interface elements, and for attaching them to the screen. It also must react to input of the user and must display the correct value in the cell.

The object must support two kind of states for the interface: One where the user can change the value in the cell, and one that is read only.

2. In the constructor functions of the `EditableCell` (*js/editableTable.js*) and `ViewCell` (*viewTable.js*) objects, add a “case” statement to the `switch`-construct. In this new branch, the correct type object for the new data type must be created.
3. Go to the `<applRoot>/save.php` file at about line 180. Check whether the new data type can be inserted using the default branch of the `switch` statement, or whether a new case needs to be created. Do the same for the `switch`-construct located around line 380 in the same file.
4. Go to the `<applRoot>/search.php` file. About line 310 a huge `switch` statement starts. Insert a new `case`-branch for the new data type.

## 9.6.2 Extended Types

### Structure of Extended Types

As mentioned above, the idea of the extended types was to provide a possibility to introduce new types without the need to change critical parts of the software. A further advantage of the chosen implementation is, that it allows easy creating of non-scalar (following the definition for scalar given above) data types.

As for the basic types, also a type object for each extended type must exist. In addition to the tasks of the type object of basic types, the type objects of extended types are responsible for loading, saving and deletion of the values. (For the basic types, this is done by the `Row` object.)

On the server side, these actions are not done by the `<applRoot>/save.php`, `<applRoot>/delete.php` or similar kinds of loading scripts. For each extended data type,

an own php file exists, that implements all these actions.

During the export process, the creation of the export values for extended type is not done directly in the `createExportXML(argsArray)` function (defined in `<applRoot>/exportFunctions.php`). For each extended type, an own function must exist that creates the export output. `createExportXML(argsArray)` only calls the `getExportData(arg1, arg2)` function (defined in `<applRoot>/exportExtendedTypes.php`). This function then passes the request further to the function that creates the export output for the extended type. This function should also be located in `<applRoot>/exportExtendedTypes.php`.

Also the call to the constructor function for extended type object is not directly in the constructor of the `EditableCell` (or `ViewCell`). Only a call to the `getExtendedType(arg1, arg2, arg3)` function (defined in `<applRoot>/js/types.js`) is performed. There the correct type object is created and returned.

One disadvantage of extended types is, that it is yet not possible to define a search constraint for them. So, no constraints for values of extended types can be made in search requests.

At the basic types, the value of the `<shortName>` child element of the `<attribute>` element was used to identify the attribute uniquely. It was therefore used as name of the database table field too. The individual type was defined by the `<type>` element.

For the extended types, the value of the `<type>` element is always the same (“extended”). Therefore, the value of the `<shortName>` is not only the identifier for the attribute, but also defines the extended of type the attribute. This value is used in the `getExtendedType()` function and the `getExportData()` function to distinguish between the different extended types.

If the same extended type is used for more than one attribute, there’s no need to repeat the implementation of the whole data type. Only a list of case statements (`case "type1": case "type2" ... case "typen":`) is required within these two functions to map all attributes with the same extended type to the same implementation.

### **Adding/Removing an Attribute with Extended Data Type:**

The following steps are necessary to add a new extended data type:

1. Create a new type object, which implements the interface described below and which implements the other requirements given below.
2. Go to `<applRoot>/js/types.js` and alter the `getExtendedType(arg1, arg2, arg3)` method:  
Add a new branch to the “switch-statement”. This `case` shall be true if the first argument equals the name of the new extended type. Within this `case`, a new instance of the the type object responsible for the new data type must be created and returned.
3. Create a new php file in `<applRoot>`. In this file, implement all scripts necessary to save, load and delete data of the new type. This file is called directly from the type object defined in step 1 and instantiated in step 2.
4. Go to `<applRoot>/exportExtendedType.php` and create a new function. It should be named following the scheme `get{nameOfExtendedType}Content(arg1[, ...])`.



This function must load all values of the specified attribute for the given row (defined by the first argument). It should create an XML formatted string out of these values and return this string. (See function `getScreenshotContent(techId)` for an example.)

5. Go to `<applRoot>/exportExtendedTypes.php` and alter the `getExportData(arg1, arg2)` function:  
Add a new “**case**”-branch to the **switch**-statement. This case shall become active if the value of the second argument of the function equals the name of the new extended type. If this case statement is executed, the function defined in step 4 shall be called. The value returned by this function shall be returned to the caller of the `getExportData(arg1, arg2)` function.

If the defined extended data type is used for more than one attribute, don’t forget to repeat steps 3 and 5 for each attribute with this type.

**Type Object for an Extended Type** An object that shall be used as “type object” for an extended data type must perform the following tasks:

Like all type objects, it is assigned to a cell object. For this cell object, it is responsible to load the data from the server. This must be done when it is created. It is also responsible for deletion of the values. If the value of its **specialSave** (see interface description below) field is true, it must also save the data to the server (otherwise this is done by the **Row** object to which the corresponding cell object belongs).

As type objects for basic type values, this object must react to inputs of the user and must support two kind of states for the interface: One where the user can change the value in the cell, and one that is read only.

A type object for an extended type must also implement the following interface:

- **int compare(Cell myCell, Cell otherCell):**  
This method takes two **Cell** objects as arguments. It returns -1 if the value of the first cell is smaller than the value of the second cell, 0 if the values of the cells are equal, and 1 if the value of the first cell is bigger than the value of the second cell.
- **void deleteEntry(int id):**  
This method deletes all values of the attribute for the data entry defined by the given “id” argument.
- **void save():**  
This function shall save the current value of the corresponding cell to the permanent storage. This function is called by the **save()** method of the **Row** object the cell belongs to. If the value of the cell is saved else when and not when this call happens (like for the “Screenshots” type), it can implement any functionality needed (or have an empty function body too).
- **specialSave:**  
This must be a field of boolean type that is readable from the **Row** object the corresponding cell belongs to. If it is **true**, this means that saving is done by the type object. If the value of this field is **false**, this means that the value of the cell shall be saved by the **Row** object. This assumes, that there exists a database field with the

name equal to the “shortName” of the attribute, to which the data can be written. Additionally, if `specialSave` is `false`, the value of the “shortName” of the attribute must not equal the name of any existing basic data type. Further, the `default` branches of the `switch`-statements in `<applRoot>/save.php` (in lines 180 and 380) must be sufficient for writing the data to the persistent storage.

## Predefined Extended Types

In the current version, only the extended data type “screenshots” exists. It stores screenshots. For each data entry, more than one screenshot can be saved. Each screenshot consists of:

- A name (as string)
- The id of the data row to which the screenshot belongs
- A number identifying the screenshot within the data row
- A description (a string)
- The name of the image
- Three images: one original image, one thumbnail whose longer side is reduced to 50 pixels, and one thumbnail whose longer side is reduced to 100 pixels

All information is stored in a database, except the images which are stored separately.

For an EER diagram of the database used to store the screenshots please see Figure 13.1.1 in the Appendix. An overview of the database table used to store the data is given here (Table 2).

name	values/type	constraints	comments
number	INT(10)	NOT NULL	PRIMARY KEY (with techId)
techId	INT(10)	NOT NULL	PRIMARY KEY (with number)
name	VARCHAR(200)	NOT NULL	FULLTEXT index
path	VARCHAR(200)	NOT NULL	FULLTEXT index
description	LONGTEXT	NOT NULL	FULLTEXT index

Table 2: The fields of the database table used to store values of “screenshot” type.

The images are stored in subfolders of the `<applRoot>/screens` directory. Thereby, following directory structure is used:

For each data row containing a screenshot, a directory exists as direct subfolder of `<applRoot>/screens`. It’s named with the id of the entry. Within this directory, for each screenshot a further directory exists. This is named according to the number of the screenshot within the entry. Within this directory, the original image is saved under its original name. The names of the thumbnails are created according to the following schema:

- `{originalName}_preview.{type}` (the larger one)

- `{originalName}_preview.{type}` (the smaller one)

They are saved in the same directory.

Once all screenshots of a data row are deleted, the directory for the data row is removed as well.

WARNING: If running DQT under windows, there's a problem removing the empty directories. They remain on the server.

The screenshots are not defined or edited directly in the table cell, but in a popup-window that can be opened by clicking an icon. Also saving is done using this popup-window. This is why the `save()` method of the `ScreenshotsType` object is empty. The page that is shown in the popup-window is `<applRoot>/screensEditPage.php`.

The php file that is responsible for saving, loading and deletion of screenshots on the server side is `<applRoot>/manageScreenshots.php`.

When comparing two screenshot values, their number of screenshot items are compared. The one with more entries is regarded to be larger than the other.

## 9.7 Export

This sections explains how to create and add an XSLT stylesheet which implements a transformation of the export output.

### 9.7.1 Export Format Syntax

The XML output created by the export functionality of DQT follows the DTD given in listing 2.

Listing 2: DTD for the output of the export functionality of DQT

```
<!ELEMENT data (head , body)>
<!ELEMENT head (tr+)>
<!ELEMENT body (tr+)>
<!ELEMENT tr (semanticGroup+ | attributeUnit+ | attributeGroup+ |
    attribute+ | value+)>
<!ATTLIST tr
    type (sgHeaderRow | auHeaderRow | agHeaderRow | aHeaderRow |
        dataRow) #REQUIRED
    entryId CDATA #IMPLIED
    entryName CDATA #IMPLIED
>
<!ELEMENT semanticGroup (#PCDATA)>
<!ATTLIST semanticGroup
    idName CDATA #REQUIRED
    colspan CDATA #REQUIRED
>
<!ELEMENT attributeUnit (#PCDATA)>
<!ATTLIST attributeUnit
    uId CDATA #REQUIRED
    colspan CDATA #REQUIRED
    semG CDATA #REQUIRED
```

```

>
<!ELEMENT attributeGroup (#PCDATA)>
<!ATTLIST attributeGroup
  gId CDATA #REQUIRED
  semG CDATA #REQUIRED
  semId CDATA #REQUIRED
  colspan CDATA #REQUIRED
>
<!ELEMENT attribute (#PCDATA)>
<!ATTLIST attribute
  attId CDATA #REQUIRED
  parAnonym (t | f) #REQUIRED
  dataType (fpn | int | string | md5 | stringlist | urllist |
    text | enum | set | extended) #REQUIRED
  semG CDATA #REQUIRED
  attGid CDATA #REQUIRED
  attG CDATA #REQUIRED
  semId CDATA #REQUIRED
>
<!ELEMENT value (#PCDATA | ss | v)*>
<!ATTLIST value
  semG CDATA #REQUIRED
  attGid CDATA #REQUIRED
  attG CDATA #REQUIRED
  type CDATA #IMPLIED
  att CDATA #REQUIRED
  attId CDATA #REQUIRED
  semId CDATA #REQUIRED
>
<!ELEMENT ss (name, description , path)>
<!ATTLIST ss
  number CDATA #REQUIRED
>
<!ELEMENT description (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT path (#PCDATA)>
<!ATTLIST path
  img CDATA #REQUIRED
>
<!ELEMENT v (#PCDATA)>

```

## Design Aspects of the Export Format

The XML format used for the export is similar to how a table is defined in HTML. It consists of a head part, which contains information about the data structure (which attributes exist and how are they structured), and a body part that contains the values of the different entries.

The structure of this XML data allows a very simple transformation of the output to any

type of table (html - table, a table defined in LaTeX, ... ).

Because DQT is supposed to deal mainly with data that can be very clearly laid out as table, this structure has been chosen and preferred over a structure like shown in listing 3.

Listing 3: Possible format for the XML output of the export function that has not been chosen

```
<data>
  <semanticGroup>
    <attributeUnit>
      <attributeGroup>
        <attribute>
          <value rowId="id1">value</value>
          <value rowId="id2">value</value>
          ...
        </attribute>
        ...
      </attributeGroup>
      ...
    </attributeUnit>
    ...
  </semanticGroup>
  ...
</data>
```

Although the data is organized as in a table, and therefore a lot of information could be retrieved from the context an element is in, the chosen XML output also provides a lot of redundant information. (For example: for each <value> element, the names and ids of the attribute, the attribute group and column group it belongs to are given explicitly. Each of these information could be retrieved out of the structure of the document.)

The reason for this redundant information is, that the XML document is transformed by XSLT. Retrieving all these information during the transformation process would make the transformation very slow. This is because the XSLT processors are not optimized to select specific information out of a file. To speed up the transformation, several information is therefore stored locally in the element, so that the XSLT processor does not have to search intensively during transformation.

### 9.7.2 Export Format Semantic

In this section, an explanation of the elements that may appear in the export XML output is given.

#### Output Head

The head of the output consists of:

- One semantic group row (<tr type="sgHeaderRow"/>)
- A number n of attribute unit rows (<tr type="auHeaderRow"/>), where n is the maximum number of nested attribute units within the exported attributes

- One attribute group row (<tr type="agHeaderRow"/>)
- One attribute row (<tr type="aHeaderRow"/>)

The *semantic group row* contains one <semanticGroup> element for each semantic group exported. For each element, the *idName* attribute contains the identifier of the group. The number of attributes belonging to this semantic group which are also exported is given in the *colspan* attribute. The text content of the <semanticGroup> element is the full name/label of the semantic group.

Each *attribute unit row* contains an <attributeUnit> element for each exported attribute unit in the level of attribute units which the row represents. Before the attribute unit rows are created, the table is “completed” as described in 9.2.2 “Table Structure Definition File”. For each missing attribute unit, an “anonymous attribute unit” is inserted. The *uId* (unitId) attribute of the <attributeUnit> element must contain the identifier of the attribute unit (for anonymous attribute units an identifier is created as described in 9.2.2 “Table Structure Definition File”). The *semG* attribute contains the full name of the semantic group to which the attribute unit belongs. (It is the string defined by the name attribute of the <semanticGroup> element in the table structure XML file). The *semId* attribute contains the identifier of the semantic group. The *colspan* attribute contains the number of attributes covered by this attribute unit. The text content of the <attributeUnit> element is the full name/label of the attribute unit (empty if anonym).

The *attribute group row* contains one <attributeGroup> element for each exported attribute/column group. If attributes are exported for which no attribute group has been explicitly declared, an “anonymous attribute group” is created, according to the rules described in section “Table Structure Definition File”. The *gId* attribute of the <attributeGroup> element contains the identifier of the attribute group. The *semG* attribute contains the full name of the semantic group to which the attribute group belongs to. (It is the string defined by the name attribute of the <semanticGroup> element in the table structure XML file.) The *semId* attribute contains the identifier of the semantic group. The *colspan* attribute contains the number of attributes covered by this attribute group. The text content of the <attributeGroup> element is the full name/label of the attribute group (empty if anonym).

The *attribute row* contains one <attribute> element for each exported attribute (column). The *attId* attribute of these elements contain the identifier of the attribute. The *semG* attribute contains the complete name of the semantic group to which the attribute belongs. The *semId* attribute contains the identifier of the semantic group. *attG* stores the complete name/label of the attribute group to which the attribute belongs, and *attGid* contains the identifier of this group. The *parAnonym* attribute of the <attribute> element indicates whether the attribute belongs to an anonymous column group (value = “t”) or not (value = “f”). *dataType* defines the data type of the attribute/column. The text content of the <attributeGroup> element defines the full name/label of the attribute.

## Output Body

The body of the output consists of a set of data rows. Each of these data rows contains the values of one table row. Each <tr> element in the body has three attributes:

- *entryId* contains the identifier of the row.

- *entryName* contains the complete name/label of the row.
- *type* contains the string “dataRow”.

Each data row contains one `<value>` element for each exported attribute. Each of these `<value>` elements possesses the following attributes:

- *semG* contains the full name/label of the semantic group to which the value belongs.
- *semid* contains the identifier of the semantic group.
- *attG* contains the full name/label of the attribute group to which the value belongs.
- *attGid* contains the identifier of the attribute group.
- *att* contains the full name/label of the attribute the value describes.
- *attId* contains the identifier of the attribute.
- *type* defines the data type of the value.

The content of the `<value>` element depends on the data type:

*int, string, text, enum, fpn, md5:*

If the attribute is of one of these data types, the value is simply added as text content to the `<value>` element.

*enum:*

If the attribute is of type “enum”, the label of the selected entry is written to the output.

*fpn:*

If the attribute is of type “fpn”, one of the characters ‘f’, ‘p’ or ‘n’ is written to the output.

*set, stringlist, urllist:*

For attributes of one of these data types, the selected entries (“set”-type) or the list items are not directly added to the content of the `<value>` element. For each entry or list item an element `<v>` (abbr. for value) is added as child element to the `<value>` element. The content of these `<v>` elements is the label for the entry (not the identifier) for the set type, and the list entry defined by the user for one of the list types.

For example, the output of the list stored as “item1;item2” is

“`<value ...><v>item1</v><v>item2</v></value>`”.

*screenshot:*

. The output for screenshot values is more complex. For each screenshot, a `<ss>` (Screen-Shot) element is appended to the `<value>` element. The *number* attribute of this element holds the number which identifies the screenshot uniquely within the entry.

The child element `<name>` contains the name of the screenshot as defined by the user. The `<description>` element contains the text defined by the user to explain the screenshot, and the `<path>` element contains the name of the original size image (with type ending).

### 9.7.3 Export Stylesheets

The XML output described in the previous section can be transformed using an XSLT stylesheet.

These stylesheets are stored in the `<applRoot>/XMLFiles/exportStylesheets` directory. When the `<applRoot>/viewPage.php` file is called, each file present in this directory is

automatically added to the list of available transformations. Then the user can chose one out of it.

To add (or remove) a transformation, only an XSLT file implementing the transformation needs to be added (or removed) to (from) this directory.

**WARNING!!** The application does not check whether the files in this directory are “\*.xsl” files or whether they are valid XSLT files. All files in the directory are added to the list of available transformations. If a file is not a valid XSLT file, trying to use this file for transformation will lead to an error. Instead of the transformed output, the error message created by the XSLT processor will be displayed. (Make therefore always sure this directory only contains proper XSLT files.)

**HINT:** There is a maximum time a php script is allowed to run on the webserver (defined by the `max_execution_time` value in the `php.ini`). If this time is exceeded by a script, it is killed. Because the XSLT transformation is done within a php script, the transformation time must be less than this specified maximal runtime.

As XSLT transformations often are quite time consuming, the standard value for this timespan (30 seconds) may be too small.

## 9.8 Configuration File `<applRoot>/conf/config.php`

This file stores all parameters and values necessary to configure/customize DQT. It is included by all server script files.

To distinguish between configuration variables and other variables as well as to avoid name clashes, all variable names defined in the *config.php* file, and only these names, start with the prefix “`conf_`”.

The only exception from this rule are the four values that are needed to connect to the database, whose names start with the prefix “`db_`”.

A detailed description of the values that can be defined in this file is given by the inline documentation of the file and in the section 9.9 “Managing Different Data Collections”.

## 9.9 Managing Different Data Collections

The XML Table Query Editor supports the management of different data collections within the same installation of DQT. (For example: in the installation containing the data collection about the techniques for visualization of temporal data, besides this data also the data about existing users for the installation can be handled.)

### 9.9.1 Handling of Different Data Collections within one DQT Installation

When calling a server page (*editPage.php*, *viewPage.php*, ...) or a server script (*save.php*, *delete.php*, ...) DQT expects an URL parameter `cfg_pref` (abbr. for “configuration prefix”). This is supposed to have an integer value  $\geq 0$ . If it is not defined, the default value is 0. This value defines the data collection that shall be accessed by the call. (For example, in the installation about techniques for visualization of temporal data, `cfg_pref` = 0 as default value is assigned to the data about visualization techniques, and `cfg_pref` = 1 is assigned to the data for user management.)

In the `<applRoot>/conf/config.php` file, which is included by all server script files (see 9.8



“Configuration File `<applRoot>/conf/config.php`” or its inline documentation), all configuration values, which are dependent of the data collection to access, are set according to the value of the `cfg_pref` URL parameter.

Currently, these are the following values:

- `$conf_nameTable` and `$conf_nameField`: They define the field in the database that stores the name of a row (the name that is displayed in the first cell of each row).
- `$conf_defaultClientOrderSG` and `$conf_defaultClientOrderAtt`: They define the attribute that shall be used for the default table order. The first one defines the id of the semantic group. The second defines the attribute within this group. This uniquely defines one attribute.
- `$conf_defaultOrder`: It defines the field that shall be used to sort the result of the `SELECT` statement on the database.
- `$conf_ts`: Stores the name of the XML file containing the table structure of the current data collection. The value must be defined as “*name.xml*”. The file must be stored in `<applRoot>/xmlValues`. (See 9.2.2 “Table Structure Definition File”.)
- `$conf_ts2editTs`: Stores the name of the XSLT file that shall be used to filter those columns from the table structure XML file, which shall be displayed on the edit page. (See section 9.4 “Defining the Attributes to be Displayed”.)
- `$conf_ts2viewTs`: Has the same function as `$conf_ts2editTs`, but only for the view page. (See section 9.4 “Defining the Attributes to be Displayed”.)
- `$conf_lastUpdateTable` and `$conf_lastUpdateField`: They define the database table and field that stores the date of the last update of each entry. This must be a field of type `TIMESTAMP`. If these values are defined, each update of an entry in the database sets the value of this field to `now()` ;.
- `$conf_edit_loadTable`: Stores the URL of the file that shall be used for loading the table structure on the edit page. The URL must be given relative to `<applRoot>` without leading “/”.
- `$conf_edit_save`: Stores the URL of the file that shall be used for saving values to the database. The URL must be given relative to `<applRoot>` without leading “/”.
- `$conf_edit_loadData`: Stores the URL of the file that shall be used to load from the server on the edit page. The URL must be given relative to `<applRoot>` without leading “/”.
- `$conf_edit_delete`: Contains the URL of the file that shall be used to delete data from the database. The URL must be given relative to `<applRoot>` without leading “/”.
- `$conf_view_loadTable`: Stores the URL of the file that shall be used for loading the table structure on the view page. The URL must be given relative to `<applRoot>` without leading “/”.

- **\$conf\_view\_export**: Contains the URL of the file to use to export data. The URL must be given relative to *<applRoot>* without leading “/”.
- **\$conf\_searchData**: Defines the URL of the file to use to search for entries. The URL must be given relative to *<applRoot>* without leading “/”.
- **\$conf\_showDetailsLink**: This field defines whether the name of rows shown in the manage cells shall be linked to the “details page” or not.

For more details see the inline documentation of the *<applRoot>/conf/config.php* file.

The values of the files storing the insert, delete, ... scripts will probably be the same for most data collections, with only a different value for the **cfg\_pref** URL parameter.

The *config.php* file further stores the value of the **cfg\_pref** URL parameter to a session variable named “cfg\_pref”. It can therefore be accessed from all server scripts. (Because the *config.php* file is called whenever a php file on the server is accessed by the client, the value of the “cfg\_pref” session variable is rewritten with every call from the client. It is therefore not capable of remembering the value from one request to another. Therefore, the **cfg\_pref** parameter must be set for each request. If it is not set, the default value is used.)

Besides the *config.php* file, only the user objects may have to be changed for introducing a new data collection. On the server as well as on the client, the user objects provide methods that return whether the current user has a certain privilege or not. Because a user may have different privileges on different data collections, the return values of these methods also depend on the current value of **cfg\_pref**.

On the server, each of these methods expects the **cfg\_pref** value as last parameter. If it is not defined, the 0 (the default value) is applied.

On the client, the **cfg\_pref** value is passed as argument to the constructor function of the user object, and is stored in the **mode** field of the user object. When one of the methods mentioned above is called, the value stored in this field is used to return the correct value. The user objects are defined in *<applRoot>/userObject.php* for the server, and *<applRoot>/js/indexJS.js* for the client.

## 9.9.2 Adding a new Data Collection

When adding a new data collection, following steps have to be performed:

1. Create a new XML file defining the table structure for the new data. (See section 9.2.2 “Table Structure Definition File”.) Also create the corresponding database table(s) to store the data in. (See section 9.3.1 “Mapping of the Table Structure XML File to the Database”.)
2. Define a new value for the **cfg\_value** for this data collection.
3. Make sure that the edit and/or view page for the new data collection is called with the **cfg\_pref** URL parameter set to the correct value. (For adding the link to the navigation bar, change the **getNavigationBar()** method of the **userObject** in the *<applRoot>/userObject.php* file.)

4. Extend the `switch($cfg_pref)` statement in the `<applRoot>/conf/config.php` file for a `case` that catches the `cfg_pref` value of the new collection. Define the values described above (and already defined in the default case) according to the new data collection.  
(When defining the URLs for script and loading server files, remember that they must not start with “/”, and that they must already contain the proper `cfg_pref` URL parameter.)
5. Change the user objects (in `<applRoot>/userObject.php` and `<applRoot>/js/indexJS.js`), so that all methods, whose output or return values depend on the `cfg_pref` value, return/output the correct values.  
This step is only necessary if the rights needed to perform an operation on the new data collection differs from the rights required to perform the same operation on the data collection assigned to `cfg_pref == 0`.

## 9.10 User, User Rights and User Related Configurations

DQT includes a simple right management system which can be used to restrict access to certain functionalities of the application. Therefore, different users can be created, and different privileges can be granted to them.

A list of the privileges that can be granted to a user is given in section 9.10 “User Rights”. Each user then has the combination of all rights/access granted to her/him. DQT allows all combinations of privileges to be given to a user. But not all possible combination of user rights are useful, as some rights require the user to possess other privileges. It’s left to the one who grants rights to other users to only assign combinations of privileges that provide the user with all intended rights.

### 9.10.1 User Rights

This section lists the existing user rights and explains the privileges needed for specific action. In the following, besides the fact that a user is allowed to open the according page, “access to the x- page is granted” or “provides access to the x- page” means, that a link to this page is shown in the navigation bar.

#### **read**

The user has only the right to view the data. One therefore has access to the view page, but not to the edit page. On the view page, the view table is displayed as well as the table with the hidden rows and the area with the display controller checkboxes for the view tables.

On the edit page, this right affects that the data rows are loaded into the edit tables. But it requires at least one out of the further rights “insert”, “edit”, “delete” to display these tables.

#### **insert**

It provides the right to insert new rows/data entries. It does not allow to view existing entries, neither to edit nor to delete them. It only provides access to the edit page, not to the view page.

If the user possesses this right only, no row from the database is loaded into the edit tables. However, the user can add rows, edit their values and save them to the database. But once a row has been saved, the user is not allowed to change the values of this row any more, nor to delete this row again. The row stays visible in the edit table, but it's impossible for the user to save the row values.

On the edit page, this right shows the (empty) edit tables, and activates the button for adding new rows. It also shows the save button in new data rows.

The “insert” right has no effect on the view page.

## **edit**

This provides the right to change values of already existing rows/data entries, and therefore only grants access to the edit page. The right to edit data does not imply the right to read/view existing data as well. If a user has the “edit” right, but not the “read” permission, no data is loaded on the edit page, so there's nothing to edit for the user. Therefore, the read right must be assigned to the user as well.

This right also does not allow to insert new data rows, so the button for adding new rows is not displayed on the screen.

If this right is only combined with the insert right, then the user still can edit the entries s/he inserted after they were saved for the first time.

Deleting existing entries from the database is also not included in this right, so the rows shown on the edit page do not provide the “delete button”.

On the edit page, this right shows the (empty) edit tables, as well as the “save” buttons in data rows loaded from the server.

The “edit” right has no effect on the view page.

## **delete**

It provides the right to delete existing rows/data entries from the database, and therefore grants access to the edit page. This right does not include the right to read/view the data as well. So if the read right is not explicitly assigned to the user, no data row is loaded and shown. So the user has no possibility to delete any row.

Therefore if the user shall be able to delete rows from the database, s/he needs at least the “delete” and the “read” right.

If this right is combined with the “insert” right only, then the user can delete the entries made by him/her after the first saving.

On the edit page, the “delete” right shows the (empty) edit tables, as well as the delete buttons in all displayed data rows.

It has no effect on the view page.

## **search**

It provides the right to use the search functionality of the application. If this right is assigned, the search part of the view page (see the user manual for details) is shown, and the user is allowed to define and submit search requests. The “search” right therefore provides access to the view page.

This right does not imply the read right. As using the search functionality without the right to see the results would not make sense, it is strongly recommended to grant the “read” right to the user as well. If only the “search” right is granted, the search functionality will

not work properly (e.g., there won't be the possibility to choose the columns for the search tables).

## **export**

It provides the right to use the export functionality of the application. That means to output the data shown in the view table after performing a transformation on it. It grants access to the view page. On the view page, this right is responsible for loading the “export area” (see the user manual).

This right does not apply the “read” right implicitly. This must be explicitly granted to make the export functionality work. If the user only has the “export” right, the UI elements for the export are loaded, but no export can be done. (This is because without “read” permission, there's no view table and without view table there's no data that could be exported.)

It is therefore strongly recommended to always combine the “read” right to the export permission.

## **users**

It provides the right to manage the user rights (create-, delete users, set passwords, assign rights).

In the user edit-mode (see 9.9 “Managing Different Data Collections” for details), this right grants access to the edit page, and includes the “read”, “insert”, “edit” and “delete” rights for this mode.

It does not grant any right in the “normal” mode.

### **9.10.2 Default (guest)**

To users that are not logged in (“guest”), only the read right is applied.

The set of default rights is defined in the `<applRoot>/conf/config.php` file by the value of the `$conf_defaultRights` field. It contains an array of boolean of length 7. Each entry in this array defines whether a certain right is applied to the default user. The following list shows the corresponding indices in the array:

index: name of right (value for guest user)

- 0: insert (false)
- 1: edit (false)
- 2: delete (false)
- 3: export (false)
- 4: read (true)
- 5: user (false)
- 6: search (false)

As an example, in the current version the declaration of this array is:

```
$conf_defaultRights = array(false,false,false,false,true,false,false);
```

If a user logs in, these default rights are completely overwritten with the rights of the user. So if the “read” right is not explicitly granted to the user, it does not have it as long as s/he's logged in, although s/he has it while s/he's logged out.

### 9.10.3 Management and Storage of User Data

All information about existing users are stored and can be managed using the data management function of DQT. (Except the default/guest user, whose rights are implemented as described above.)

The table structure of the user data is defined by the following XML file:

*<applRoot>/XMLData/tableStructure\_users.xml*):

Listing 4: XML file defining the structure of the user data

```
<?xml version="1.0"?>
<attributes>
  <semanticGroup name="user information" tName="users">
    <attribute>
      <name>id</name>
      <shortName>id</shortName>
      <type>int</type>
    </attribute>
    <attribute>
      <name>login</name>
      <shortName>login</shortName>
      <type>string</type>
    </attribute>
    <attribute>
      <name>password</name>
      <shortName>password</shortName>
      <type>md5</type>
    </attribute>
    <attribute>
      <name>rights</name>
      <shortName>rights</shortName>
      <type>set</type>
      <value>
        <name>insert</name>
        <shortName>insert</shortName>
        <type>jn</type>
        <order>2</order>
      </value>
      <value>
        <name>edit</name>
        <shortName>edit</shortName>
        <type>jn</type>
        <order>3</order>
      </value>
      <value>
        <name>delete</name>
        <shortName>delete</shortName>
        <type>jn</type>
        <order>4</order>
      </value>
    </attribute>
  </semanticGroup>
</attributes>
```

```

    </value>
    <value>
      <name>export </name>
      <shortName>export </shortName>
      <type>jn </type>
      <order>6</order>
    </value>
    <value>
      <name>user </name>
      <shortName>user </shortName>
      <type>jn </type>
      <order>7</order>
    </value>
    <value>
      <name>search </name>
      <shortName>search </shortName>
      <type>jn </type>
      <order>5</order>
    </value>
    <value>
      <name>read </name>
      <shortName>read </shortName>
      <type>jn </type>
      <order>1</order>
    </value>
  </attribute>
  <attribute>
    <name>last Update</name>
    <shortName>lastUpdate </shortName>
    <type>timestamp</type>
  </attribute>
</semanticGroup>
</attributes>

```

(For a more detailed database description please see the appendix.)

If a user has the “users”-right, the edit page for the user data can be accessed by `<applRoot>/edit.php?cfg_pref=1`. (This is by calling the edit page with the value 1 for the `cfg_pref` URL parameter. A corresponding link is shown in the navigation bar (labelled with “edit users”). For a description how to use this page, see the user manual.

#### 9.10.4 Navigation Bar

Because the content of the navigation bar depends on the logged in user, the content of this part of the screen is created on the server by the user object (defined in `<applRoot>/user-Object.php`). It can be retrieved by calling the `getNavigationBar()` method of the object instance representing the current user. This instance is stored in the `$_SESSION[“user”]` variable. Depending on the user rights, links to the different parts of the page are added to the bar.

Depending on whether the user is logged in or not, the login form or the logout button is added.

To change the content or behaviour of the navigation bar, the output of this method must be changed. (For details, please see its source code documentation.)

## 9.11 index.php

`<applRoot>/index.php` is the home- and start page of DQT. Besides the navigation bar at the top of the page, in the standard installation it contains a short text congratulating the user for successfully installing DQT. To change this welcome text, the `<applRoot>/index.php` file needs to be changed. Almost at the end of the file, there's a section enclosed by the comments `<!-- THIS IS THE ACTUAL CONTENT OF THE SITE: -->` and `<!-- END OF THE SITE CONTENT -->`. This area should be used to define the additional content of the start page. Please use XHTML. PHP statements are allowed.



## 10 User Manual

The following manual describes the use of DQT from a user account that has unrestricted access privileges. For user with fewer rights, handling of the different functions does not differ. Only some functionality will not be available to users with restricted access. It is pointed out which rights are needed by the user to access a certain function or what happens if a user does not have the necessary privilege.

### 10.1 Starting DQT

The DQT has two different windows: The main window and the detail- or entry- window. To access the main window (where editing, searching and exporting is done), load the URL where the instance of DQT is located in your browser (we refer to this URL as `<applRoot>` in this section; “`http://localhost/dqt/`” for example). To directly access the edit- or view-page (see sections 10.3 “Edit Page” and 10.4 “View Page” for details), append *edit.php* or *view.php* to `<applRoot>` (e.g., “`http://localhost/dqt/edit.php`”).

How to access the detail- or entry- window, please see section “detail-/entry- window”.

### 10.2 Main Window

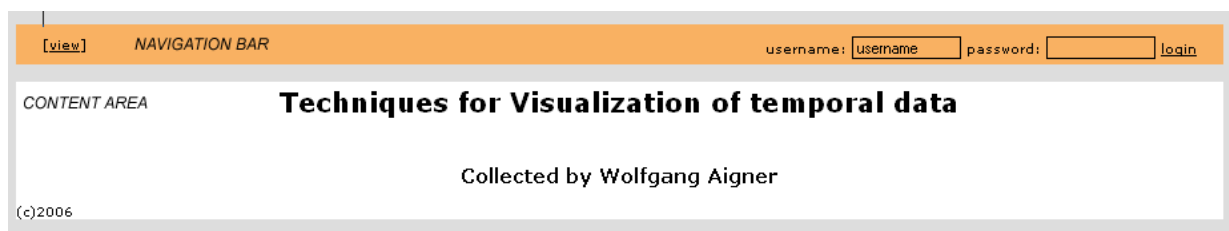


Figure 9: Main window after starting DQT using the `<applRoot>` URL.

The main window screen is divided into two sub areas (see Fig. 9):

- the “navigation bar” at the top
- the “content area” filling the rest of the screen

The display of the content area depends on the operational mode DQT is currently in. Two main modes exist:

The edit mode (also called “edit page”) and the view mode (also called “view page”).

Unlike the content area, the navigation bar is independent from the currently active application part.

#### 10.2.1 Navigation Bar

The navigation bar displays following information:

On the left side, there are links that provide access to the functionality of DQT. On the right side, information about the login status of the user as well as the login/logout fields are shown.

The content actually displayed in the navigation bar is independent of which section of the currently active part of DQT, but depends on whether the user is logged in or not, and therefore on its privileges.

**Login Information** If no user is logged in, following form is shown that can be used for login (see Fig. 9 and Fig. 10).

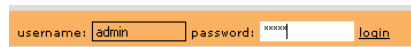
A screenshot of a login form within a navigation bar. It features two text input fields: the first is labeled 'username:' and contains the text 'admin'; the second is labeled 'password:' and contains a series of dots. To the right of the password field is a button labeled 'login'.

Figure 10: Login-form in the navigation bar

To login, the username must be filled in the text field labelled “username” (where the string “username” is filled in by default), and the password must be entered into the other text field (see Fig. 10).

There are three possibilities to start the login process:

1. By clicking onto the “login” link
2. By pressing the <alt> and the <l> key together  
(if using the Internet Explorer, <Enter> must be pressed afterwards, in Firefox this is not necessary)
3. By pressing the <enter> button while one of the two text fields is active (white background without black border)

Once the login process started, the login form is replaced by the string “logging in ...” (see Fig. 11).

A screenshot of a small orange rectangular button with the text 'logging in ...' in black.

Figure 11: During the login process, the login form is replaced by this string

If the login procedure was unsuccessful, an error message is prompted, and the “logging in ...” string is replaced again by the login form.

After successful login, the navigation bar looks similar to the one shown in Fig. 12.

A screenshot of a navigation bar after a successful login. It is an orange bar with a vertical line on the left. On the left side, there are three links: '[edit data]', '[view|search|export]', and '[edit users]'. On the right side, it says 'logged in as: admin' followed by a '[logout]' link.

Figure 12: After a successful login, this is what the navigation bar should look like

By clicking the “logout” link, or pressing <alt> + <l> together, the user is logged out. The navigation bar looks again as shown in Fig. 9.

**Links to available Functionalities** A list of links is displayed on the left side of the navigation bar. Using these links, the different functionalities of DQT can be accessed. Only links to allowed functionalities for the current user are displayed. Fig. 12 shows the links available in the standard installation for a user with unrestricted access privileges.

## 10.2.2 Content Area

As mentioned above, DQT mainly consists of the edit part (“edit page”) and the view part (“view page”). Depending on which part is currently active, the content area holds the elements to insert, edit and delete entries (edit page) or to view, search and export the data (view page).

When DQT is loaded, none of these parts is opened by default. Instead, only a short welcome message (see Fig. 9) is displayed.

Following, the edit and view part are explained in more detail.

## 10.3 Edit Page

For access to the edit page, it is necessary to have one of the following rights: “insert”, “edit”, or “delete”.

If the logged in user has at least one of these rights, a link “edit data” in the navigation bar is shown. By clicking onto this link (or pressing <alt> + <l> keys together (using the Internet Explorer <enter> needs to be pressed afterwards), the edit section (edit page) is loaded (Fig. 13).

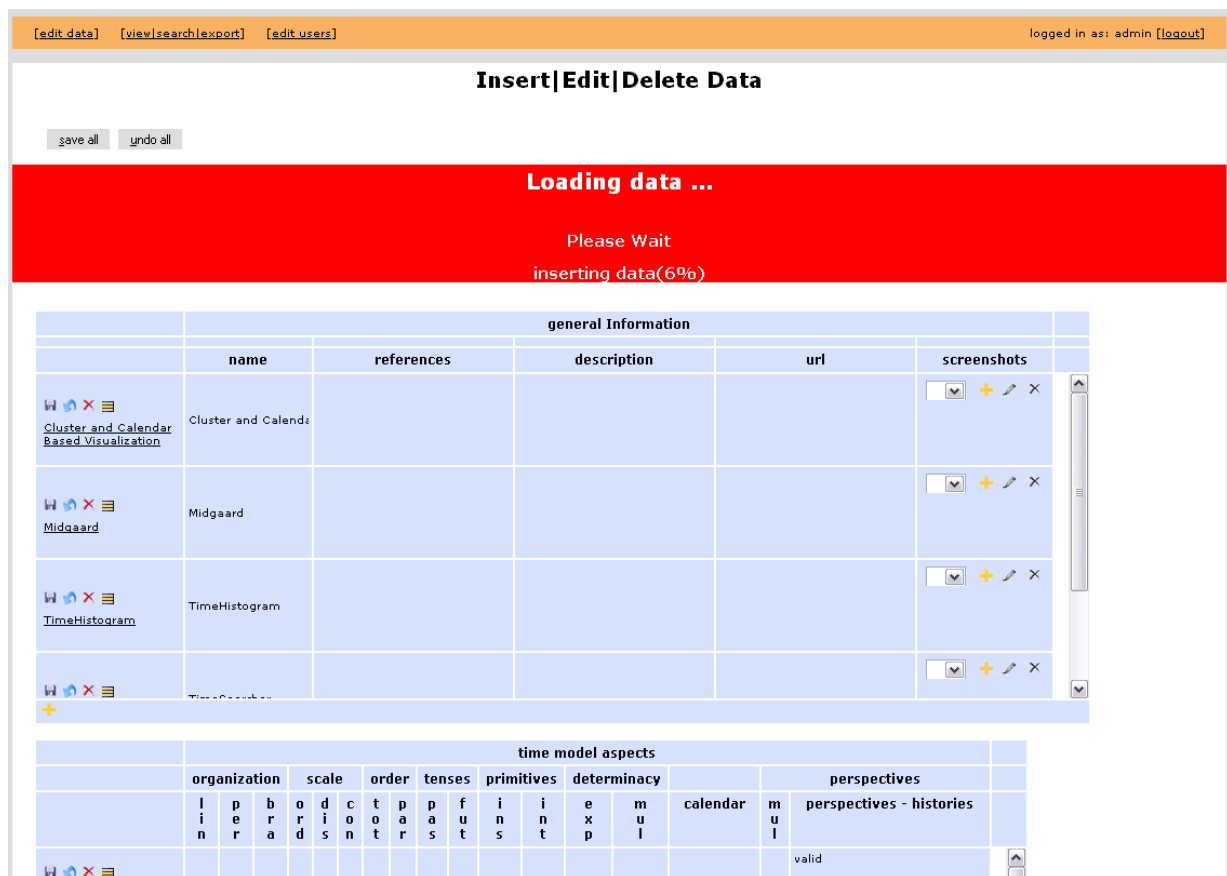


Figure 13: Screenshot of the edit page loading the data from the server

Depending of the number of entries, the loading of the data typically may take up to a few seconds. The progress is displayed within the red bar (labelled with “Loading data...”) that is closed when the data has been loaded completely. Furthermore, during data loading, all cells of the tables have a light blue background color.

Once loading is finished, the red bar is removed, and the background colour of the rows is set alternately to dark- and light- grey. The rows are listed according to the default order (Fig. 14).

general Information				
name	references	description	url	screenshots
24h Clock Display				Image 2
2D Zoom Star				Image 3
3D Spiral Display				

time model aspects												
organization				scale	order	tenses	primitives	determinacy	calendar	perspectives		
l	i	p	b	o	c	t	p	f	i	i	e	
n	e	e	r	d	o	a	a	u	n	n	x	
				s	n	r	s	t	p	m	u	
24h Clock Display	•				•	•	•	•	•		Gregorian	valid
2D Zoom Star	•	o			•	•	•	•	•		none	not fixed
												not fixed

Figure 14: The completely loaded edit page

If no data entries are displayed (for there are none stored or the user does not have the right to read existing entries), only the structure of the tables without any data rows are loaded, as shown in Fig. 15.

### 10.3.1 Edit Tables

Because the number of attributes used to classify/describe an entry may be too large to display the columns in one single line, the table (defined by the attributes and the entries) might be split into different tables (as can be seen in Fig. 14). Thereby, for each data entry, one row in each of the tables created by splitting the original table exists.

Each table consists of three main parts:

- the table head
- the table footer
- the table body





Figure 17: The header cell of a column the table is ordered by. A small arrow indicates the order direction.

## Table Footer

The table footer contains only one row (Fig. 18) with a little icon on the left side. By clicking onto this, new entries can be created (see 10.3.2 “Add New Entries”). The table footer is only shown if the user has the right to insert new entries.



Figure 18: The row in the table footer (“add row”)

## Table Body

The table body contains the data rows.

The first cell of each row is the so called “management cell” (Fig. 19), consisting of two lines. The first line contains buttons to access the functionalities of the data row. These



Figure 19: The “management cell”, the first cell of each data row

buttons are: The “save button” (see 10.3.4 “Save Changes”), the “undo button” (see 10.3.5 “Undo Unsaved Changes”), the “delete button” (see 10.3.6 “Delete Entries”) and the “highlight button” (see 10.3.3 “Highlight an Entry”). The utmost right character “\*” indicates that some value in the row has been changed since data has been loaded or saved for the last time (otherwise, the “\*” is not visible).

The second line holds the name of the entry (the row). This name is linked to the detail page (see “detail page”). It is also used to identify the row and to find all rows in different tables that belong to the same entry.

If a new (empty) entry is created, no name is displayed in the cell. The name is automatically inserted after the entry is saved for the first time.

### 10.3.2 Add New Entries


By clicking at the cursor position somewhere onto the row in the table footer, a new (empty) entry is created. That means, that in each of the tables a new empty row is inserted. This row is attached below all other rows, and all table bodies are scrolled down to this new row. The new row is only created locally on the client, but not automatically saved in the database. This needs to be done explicitly (see 10.3.4 “Save Changes”).

### 10.3.3 Highlight an Entry





















One entry can be split up to different tables on the display. This is because not all columns of the original table are displayed in one large table but are broken into different smaller tables. To easily find all rows in the different tables belonging to a certain entry, there is

a possibility to highlight all corresponding rows.

This can be accomplished in two ways:

An entry can be explicitly highlighted by pressing the highlight button  of the row in any of the tables.

Additionally, whenever one of the input elements of a cells is focused, the entry corresponding to this value or cell is highlighted.


time model aspects																	
	organization			scale	order	tenses	primitives	determinacy		perspectives							
	i	p	b	o	d	c	t	p	p	f	i	i	e	m	calendar	m	perspectives - histories
	n	o	r	d	i	s	n	a	s	u	n	n	x	p	l	u	
    *		•				•		•	•	•	•				Gregorain		
<a href="#">24h Clock Display</a>																	
    *	•	o			•		•		•		•	•			none		not fixed
<a href="#">2D Zoom Star</a>																	
   		•			•		•		•		•				not fixed		not fixed
<a href="#">3D Spiral Display</a>																	
   	•				•			•	•		•				not fixed		not fixed
<a href="#">3D ThemeRiver</a>																	
   	•				•			•	•		•						valid
<a href="#">3D ThemeRiver</a>																	

<

Figure 20: The entry "3D Spiral Display" is highlighted.

### 10.3.4 Save Changes

There are two possibilities to save changes made to an entry (or after creation of a new entry):

One possibility is to click the save button  in the first cell of each row. This button is always displayed if the user is allowed to save changes. Please note, that clicking the save button of one row does not only save the attributes of the current row, but all attributes in all displayed tables belonging to this entry.

The other possibility is to use the "save all" button, which is placed above the first table (see Fig. 13 or 14). This button can also be activated by pressing <alt>+<s> together (followed by <enter> with some browsers). This saves the changes made to all entries the user is allowed to save. If no change needs to be saved (or the user has no right so save


it), a corresponding message is displayed: “Nothing to save”.

In both cases, only those values of the cells that have been changed are sent to the server. While an entry is saved, the display of the “management cells” of all rows of this entry change. The control-buttons are replaced by a message “saving ...”. When saving has finished, the buttons are again displayed. If everything worked properly, the “\*” is removed.

WARNING! When exiting the edit page, all unsaved changes are lost. There is no automatic saving and no confirm message like “There’s unsaved data, are you sure you want to quit?”. All unsaved data-changes are simply discarded.

### 10.3.5 Undo Unsaved Changes


DQT provides the possibility to undo changes that have not been saved yet. This can be done in following two ways:

One way is to click the undo button  that is displayed in the first cell of each row. Please note, that clicking the undo button of one row does not only set back the values within the current row, but within all rows in all tables that belong to the same entry as this row.

The other possibility to undo unsaved changes is to use the “undo all” button which is placed above the first table (see Fig. 13 or 14). The button can also be activated by pressing <alt>+<u> together (followed by <enter> with some browsers). This undos all changes made to all entries.

“Undo” resets all values of an entry back to the values that are stored on the server. This means either to the values that have been loaded from the server, or to those values that have been currently saved to the server. So the “undo” action resets an entry to the last state, where there has been no “\*” displayed in the management cells of this entry. Therefore, after undoing the changes, the “\*” is no longer displayed. Applying the undo function for an entry that has not been changed (and therefore has no “\*” in its management cells) does not have any effect.

### 10.3.6 Delete Entries

To delete an entry, the delete button  of the entry must be clicked. This button is displayed for each entry the user is allowed to delete. Please note that clicking the delete button in any table not only deletes the row in this table, but also all corresponding rows in the other tables. After the button has been clicked, a confirm message is prompted whether the complete entry shall be deleted definitely.

WARNING! The entry is removed from the display and deleted from the server as well. There is no possibility of undoing this!

### 10.3.7 Different Data Type User Interfaces

Each attribute has a certain data type. This data type defines which values can be inserted into a table column. Further, each data type has its own interface, which shall help defining the value of a cell. The datatype of a column is shown as part of the information that is displayed when the cursor is moved over the header cell.



### “int”

The “int” type allows to define integer numbers between -2147483647 and 2147483647. They are inserted using a text field.



Figure 21: User interface for the “int” type: Inactive with an negative number inserted (l), inactive with a positive number inserted (c), active (r)

As long as this text field is not active, it has no border and a transparent background (Fig. 21, left and center). By clicking somewhere into the cell, the text field becomes active, showing a black border and a white background (Fig. 21, right). Now the number can be inserted. If it is tried to insert any string, that consists of characters other then “0”, “1”, ..., “9” (with an optional “-” as first character), an error message is prompted as soon as the text field loses the focus, and the value is replaced by the last saved/loaded value. The default value (the value that is inserted into a newly created cell with type int) is 0.

### “string”

The “string” type allows to define arbitrary strings with a maximal length of 200 characters. All characters supported by UTF-8 and empty strings (no character) are allowed. The strings are inserted using a text field.

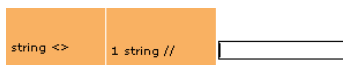


Figure 22: User interface for the “string” type: Two inactive cells with a string value (l,c) and an active cell (r)

As long as this text field is not active, it has no border and a transparent background (Fig. 22, l,c). By clicking somewhere into the cell, the text field becomes active, showing a black border and a white background (Fig. 22, r). Then the string can be typed in. If one tries to insert a string with more than 200 characters, only the first 200 characters are displayed and no more characters can be written to the text field. The default value (the value that is inserted to a newly created cell with type string) is the empty string “”.

### “md5”

“md5” is almost equal to the string type. It provides exactly the same interface as a cell of type “string”. The only difference is, that whenever the value of the cell is saved, the string is md5 hashed before it is written to the database. Thereby, DQT does not distinguish between whether a string is already hashed or whether it is plaintext, but applies the md5 hash to all strings that are saved. To avoid that an already md5 hashed value loaded from the database gets hashed again when saving the corresponding entry, no changes should be performed on this value.

## “text”

The “text” type allows to store longer string values (up to 4GByte). All characters are allowed within these strings, as well as empty strings. The text is inserted using a text-area.

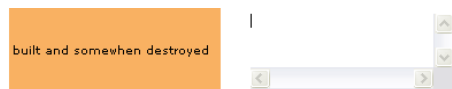


Figure 23: User interface for the “text” type: An inactive (l) and an active (r) text-area

As long as the text-area is not active, no borders and scrollbars are shown, and all text that only can be reached by scrolling is hidden. The background of the text-area is transparent (Fig. 23). By clicking somewhere into the cell, the text-area becomes active, the border and the scrollbars are shown, and the background turns into white. Then the text can be inserted. The default value (the value that is inserted to a newly created cell with type text) is the empty string “”.

## “stringlist”:

The “stringlist” type allows to define a list that contains string items. The size of the entire list can be up to 4GByte.

The “stringlist” type provides the same interface as the text type (see Fig. 23). The list is defined as semi-colon (“;”) separated list. That means, that the different list items are written as a string into the text field, and a semi-colon is used to separate two items. The list items therefore are allowed to contain all characters, except the “;”. (For example, the string “this is item1; this is item 2” would define a list with the two list items “this is item1” and “this is item 2”.) The default value (the value that is inserted to a newly created cell with type “stringlist”) is the empty string “”.

## “urllist”

The “urllist” type allows to define a list that contains string items. These string items are supposed to be valid URLs. The size of the whole list can be up to 4GByte.

The “urllist” type provides the same interface as the text type (see Fig. 23). The list is defined as semi-colon (“;”) separated list. That means, that the different list items are written as one string into the text field, and a “;” is used to separate two items. The list items are not checked whether they represent a valid URL or not. But when they are displayed in other parts of the application, each entry is displayed as a link to the target represented by the item. To create a valid link target, “http://” is added to the begin of the link, if it does not already start with this string. Therefore, currently no other protocol than http is supported. As the links are not checked, they may contain characters that are not allowed in URLs, although this is not encouraged. Only the “;” character must be avoided (as it defines the end of the list item).

For example, the string “www.tuwien.ac.at;tuwis.tuwien.ac.at” would define a list with the two list items (http://)“www.tuwien.ac.at” and (http://)“tuwis.tuwien.ac.at”.

The default value (the value that is inserted to a newly created cell with type “urllist”) is the empty string “”, and therefore the empty list.

## “enum”

“enum” is the abbreviation for “enumeration”.

The “enum” type allows to select one value out of a predefined set of values. The possible values are presented in a selection box (Fig. 24) and may depend on the attributes. Exactly one of these items must be selected.



Figure 24: Interface of the “enum” type: A drop down box, where one entry can be selected.

The default value (the value that is selected in a newly created cell with type “enum”) is the first entry in the list.

## “fpn”

The “fpn” type allows to choose between three values:

“(completely) fulfilled”, “partly or implicitly fulfilled”, “not fulfilled”. “fpn” is the abbreviation for “fulfilled”, “partly (fulfilled)” and “not (fulfilled)”. The property that is fulfilled (or not) is described by the name of the attribute. The three values are represented as shown in Fig. 25.



Figure 25: The interface of the “fpn” type: completely fulfilled (l), partly fulfilled (c), not fulfilled (r)

The value of a cell can be changed by clicking onto the cell. This changes the value from “(completely) fulfilled” to “partly fulfilled”, from “partly fulfilled” to “not fulfilled” and from “not fulfilled” to “(completely) fulfilled”. The default value (the value that is selected in a newly created cell with type “fpn”) is “not fulfilled”.

## “set”

The “set” type allows to select an arbitrary number of values from a predefined set. The selectable values are presented in a selection box (Fig. 26).

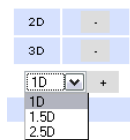


Figure 26: The interface of the “set” type: The selected values are displayed above the selection box.

To select a value, it first must be chosen from the selection box. Then, the  $[+]$  button right hand side to the selection box must be pressed (Fig. 26). The selected value is then removed from the selection box, and a new line is created above the box, consisting of the

label of the selected value and a button labelled with [-]. If there's no more entry left in the selection box, the [+] button is disabled.

To unselect an entry, the [-] of the appropriate entry must be pressed. Then the line of the entry is removed, and it is available again in the selection box.

By default (if the cell is newly created), no entry is selected.

### “screenshot”

The “screenshot” type allows to save images together with a title and a description of this image. Thereby, for each entry more than one image can be saved.

If no screenshot has been defined yet, the cell of the screenshot type looks like shown in Fig. 27.



Figure 27: Interface of the “screenshot” type if no screenshot has been defined

Only the first of the three buttons is active, and there's no entry in the selection box. To add a new image, the add button [+] must be pressed. Then a popup window opens as shown in Fig. 28.

Figure 28: The content of the popup window for inserting/editing screenshots

In the text field labelled “name” the name/title of the image can be defined. To insert a new screenshot, this text field must contain at least a string with 2 characters. The maximal length of the name/title-string is 30 characters.

The content of the text field labelled “description” can be any string from the empty string “” up to a string with a maximum size of 4GByte.

The text field labelled “picture” finally must contain the location of the image that shall be saved. By clicking the “Browse” button, an image can be chosen using a file system

browser. This value is required to create a new screenshot entry. DQT supports the image types “gif”, “png” and “jpg”. The maximum size allowed for one image is 500KByte. To save the new screenshot entry, the “save” button must be pressed. If everything works properly, the screenshot entry is stored and the popup-window closes. Otherwise, an error message is displayed and the popup-window remains open. If the “close window without saving” button is clicked, the popup closes without saving neither the values nor the image. After successful saving, an entry for the new screenshot is created in the selection box of the table cell, and the second button (the edit button) and the third button (the delete button) are enabled. (see Fig. 29)

If more than one screenshot for an entry exists, the selection box contains one entry for each screenshot.



Figure 29: Interface of the “screenshot” type with two screenshots

Below the selection box and the three buttons, a thumbnail image of screenshot currently selected is shown.

To edit or delete a screenshot, the screenshot first must be selected in the selection box. Once the desired screenshot is chosen, it can be edited by clicking the edit button or deleted by using the delete button.

When clicking the delete button, a confirmation window appears, as deletion can not be made undone afterwards.

When clicking the edit button, a popup-window (as shown in Fig. 28) opens, but with the values of the screenshot already filled in. These values can be changed and saved, or the popup-window can be closed without do any changes. Unlike when entering a new screenshot, the text field labeled with “picture” can remain empty. In this case, the existing image remains unchanged. If a new image is defined, the existing one is overwritten by the new one.

Unlike all other data types, the screenshot type is saved by clicking the “save” button in the popup-window, but not when the save button of the table row is pressed.

## 10.4 View Page

Access to the view page requires one of the following permissions: “read”, “search”, or “export”.

If the current user has at least one of these rights, a link “[view|search|export]” is shown in the navigation bar (see Fig. 12. By clicking onto this link, or pressing the <alt>+<v> button together (using the Internet Explorer <enter> must be pressed afterwards), the view part (view page) is loaded.



The content area of the view page consists of three important sections:

- the search area
- the view area
- the export area

Which of these areas is accessible depends on the individual permission of the current user.

Figure 30: The three sections of the view page: Search-, View- and Export area

Each of these areas can be expanded or collapsed, depending on whether the content of the area needs to be visible or not (Fig. 30).

To expand an area, the corresponding  button on the left side of the screen must be clicked. When an area is expanded, the collapse button  is shown instead of the expand button. By clicking this button, the area is collapsed again. When the view page is loaded, the search and view areas are expanded whereas the export area is collapsed.

#### 10.4.1 View Area

For access to the “view area”, the user needs the “read” permission.

The view area itself is divided into three subareas, which again can be expanded or collapsed (Fig. 32):

Figure 31: The three areas of the “view” area

At the top of the view area, there are tables with checkboxes to control the visibility of the columns in the view table.

The second subarea is the “view table”.

At the bottom of the the “view area”, there is the “hidden table” (see section “Hidden Table”).

When the view page is loaded, only the subarea of the “view table” is expanded. After the elements of the page are loaded, the data rows are loaded from the database. This may take some time, depending on the number of entries. The current progress is displayed in the red bar above the “view area” (Fig. 32).

Furthermore, as long as data is loaded, all cells of the tables have a light-blue as background color. After loading finished, the red “loading” bar vanishes, and the background

loading data ...														
inserting data(19%)														
view:														
select the attributes that shall be displayed														
search results:														
general Information														
unit1														
domain														
organization														
scale														
order														
tenses														
primitives														
determinacy														
calendar														
name														
references														
description														
url														
screenshots														
<input type="checkbox"/> <a href="#">Cluster and Calendar Based Visualization</a>	Cluster and Calendar Based Visualization					<input type="checkbox"/>								

Figure 32: Screenshot of the view area while DQT loads data from the server.

colour of the rows of the tables is set alternately to dark- and light- grey. The rows are listed according to the default order (Fig. 33).

view:														
select the attributes that shall be displayed														
search results:														
general Information														
unit1														
domain														
organization														
scale														
order														
tenses														
primitives														
determinacy														
calendar														
name														
references														
description														
url														
screenshots														
<input type="checkbox"/> <a href="#">3D Spiral Display</a>	3D Spiral Display					<input type="checkbox"/>								not fixed
<input type="checkbox"/> <a href="#">3D ThemeRiver</a>	3D ThemeRiver					<input type="checkbox"/>								not fixed

Figure 33: The view area after loading of the data has completed.

## View table

The “view table” is the central element of the view page. It displays all the information stored on the server, whereby the user can control the desired output. Unlike the edit page, all attributes that are used to describe/classify the different entries are shown in one table, and are not split to different tables. This is done to provide the user a better data-overview, and because the visualization of the table can be adopted to the users needs.

After initial loading, the “view table” contains one row for each entry saved on the server. The user can (to some extent) control the structure of the table, by defining which columns shall be displayed (see “Hide/Show Columns”) and by removing rows from the table (see “Hide/Show Rows”). Using the “search” functionality, individual filters can be defined for the displayed rows, and the current status of the view table can be exported using the “export” function.

**Table Structure** The “view table” consists of the table head and the table body.

time model aspects														
domain														
organization			scale		order	tenses		primitives		determinacy		perspectives		
l i n	p e r	b r a	o d c	branching type: fpn	t o t	p a r	p a s	f u t	i n s	i n t	e x p	m u l	calendar	m u l
perspectives - histories														

Figure 34: An example for the head of a ”view table”

The rows in the table head have a light-blue background color. The last row of the table head contains one cell for each column. Those cells contain the names of the attributes. By spanning over more than one column, the cells in the other rows define groups the columns belong to. This defines a structure and creates a context for the attributes and therefore provides additional information about them. For the following header cells, additional information is shown in a small pop-up area at mouse position when the mouse is moved over them: Cells that are not empty (that is, they contain a name/text) that are placed either in the first, the last, or in the row before the last row of the table head (in Fig. 34 the cursor is over the “bra” column). The table body contains one data row for each entry. The first cell of each row (“management cell”) contains a checkbox and the name of the entry (Fig. 35).

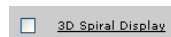


Figure 35: An example for a ”management cell” in the ”view table”

The name of the entry is linked to the detail page of the entry (see section “Detail-/Entry Window” for information about the detail page).

## Hidden Table

The “hidden table” is displayed below the view table and is collapsed when the page is loaded. The user can explicitly chose rows s/he wants to hide from the “view table”. To provide the possibility to fade in these rows again, these rows are displayed in the “hidden table”. Because this table contains the “hidden rows”, it’s called “hidden table”. For details see “Hide/Show Rows”.



## Column Control Tables

The first sub area of the view area contains those elements that control the visibility of the columns in the view table. It is collapsed by default. For details see “Hide/Show Columns”.

### 10.4.2 Sorting the Table

By clicking onto an attribute cell (a cell in the last row of the table header), the rows of the table are ordered ascending according to the column values of the attribute. Clicking again onto the same attribute cell changes the order from ascending to descending. A little orange arrow in the attribute cell shows the current order direction of the table. (Fig. 36)



Figure 36: The header cell of a column the table is ordered by. A small arrow indicates the order direction.

Depending on the number of rows and columns, the sorting of the rows may take one or two seconds. Please wait until the table is reordered before clicking again onto any cell.

### 10.4.3 Hide/Show Rows

It is possible to fade out (hide) some rows out of the view table, and to fade them in (show) again afterwards. Two possibilities (a and b) exist, each requiring two steps:

1. Check the checkboxes of the rows that:
  - (a) shall be removed from the view table
  - (b) shall remain visualized in the view table
2. Below the view table, four buttons are placed (see Fig. 37). Click the “with selected:”
  - (a) “hide” button to fade out the rows with the checked checkboxes
  - (b) “hide others” button to fade out the rows with the unchecked checkboxes

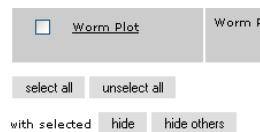


Figure 37: The buttons for selecting/unselecting and hiding rows of the “view table” (placed below the “view table”)

The buttons “select all” and “unselect all” can be used to select or unselect all checkboxes of the rows in the view table.

The “management cells” of the rows that have been removed from the view table are now displayed in the hidden table.

The rows in the “hidden table” are displayed in the same order as they have been in the “view table”. If more rows are added later to the hidden table, they are inserted above



Figure 38: The area of the “hidden table” with three hidden rows

the entries already present in the “hidden table”. If the view table is reordered, the order of the entries in the hidden table does not change.

To re-insert (show) the hidden entries to the “view table”, check their checkboxes and then one of the “show selected” buttons must be clicked. To show again all hidden entries, one of the “show all” buttons can be used.

The rows are inserted in the correct order in the view table according to its current order. Thus the view table is kept in correct order.

If after re-inserting there’s no entry left in the “hidden table”, the four buttons (2x “show selected” and 2x “show all”) become inactive, as shown in Fig. 39.



Figure 39: The area around the “hidden table” (with no “hidden row”).

#### 10.4.4 Hide/Show Columns

The visibility of the columns (in the “view table”) can be controlled using checkboxes. These checkboxes are placed in the first subarea of the view area (Fig. 40). If the controlling checkbox is checked, the column is shown, if not, the column is hidden.

Because a simple list of checkboxes labelled with the names of the columns they control might become too long to be useful, the control elements are structured in tables (“control table”). Thereby the same nesting of attributes as in the head of the view table is used (but only the highest and the two lowest levels are regarded).

For each element in the lowest level of attribute/column groups, one table row with two cells for a control table is provided. If the corresponding attribute group has a name (the cell of the group in the head of the “view table” contains a text), a checkbox labelled with this name is displayed in the first cell (otherwise this cell is empty). In the second cell, all checkboxes are placed controlling the corresponding columns. When the status of the checkbox of the group is changed, these column states are all set to the same value. The rows are organized in tables. For each group at the highest level of attribute-/column groups, there exists an own table. This table contains the rows of all low level groups belonging to this group. In the first row, there’s a checkbox labelled with the name of the group. When the status of this checkbox is changed, the state of all control elements in the table are set to the same value.

**view:**

☐ select the attributes that shall be displayed

☒ general Information  
☐ time model aspects  
☐ data aspects  
☒ representational Aspects  
☒ supported tasks and application areas  
☐ classification

<input checked="" type="checkbox"/> general Information		<input checked="" type="checkbox"/> representational Aspects		<input checked="" type="checkbox"/> supported tasks and application areas	
<input checked="" type="checkbox"/> name	<input checked="" type="checkbox"/> type	<input checked="" type="checkbox"/> static	<input checked="" type="checkbox"/> supported tasks <div style="margin-top: 10px;"> <input checked="" type="checkbox"/> existence of a data element  <input checked="" type="checkbox"/> temporal location  <input checked="" type="checkbox"/> temporal interval  <input checked="" type="checkbox"/> temporal texture  <input checked="" type="checkbox"/> rate of change  <input checked="" type="checkbox"/> sequence  <input checked="" type="checkbox"/> synchronization           </div>		
<input checked="" type="checkbox"/> references		<input checked="" type="checkbox"/> dynamic			
<input checked="" type="checkbox"/> description		<input checked="" type="checkbox"/> dimensionality			
<input checked="" type="checkbox"/> url		<input checked="" type="checkbox"/> heterogenous			
<input checked="" type="checkbox"/> screenshots		<input checked="" type="checkbox"/> interactivity	<input checked="" type="checkbox"/> application area		

Figure 40: An example for the control tables of the “view table”

For each control table, an additional checkbox is provided in the list above the display area of the control tables. This checkbox controls whether the corresponding control table is displayed or not. If a control table is hidden, the state of the columns controlled by this table is not changed.

Depending on the number of columns and rows displayed in the “view table”, it may take a few seconds to hide a column. So especially when working with larger tables, DQT should be given enough time to perform the hide/show operation after clicking onto a checkbox before the state of any other checkbox is changed.

#### 10.4.5 Search Area

DQT enables the user to search for entries in the database. Thereby the user can define constraints for the values of attributes the entries looked for must fulfill. Only those entries that fulfill these constraints are shown. The user can choose whether he wants to start a search explicitly, or whether the search shall be performed automatically whenever s/he changes the search constraints (see “automatic search” for more information).

#### Defining Search Requests

The search constraints are entered in the “search area” (Fig. 41). This is done using “search tables”. Those “search tables” are displayed at the bottom of the search area (labelled with “B” in Fig. 41). They are created using the elements at the top of the search area (labelled with “A” in Fig. 41). How this is done see “Creating search tables”.

Into these “search tables”, the values the requested entries shall fulfill are entered. DQT then returns all entries saved on the server that match the given constraints. Following rules are applied to match the search tables and their content against the entries stored in the database:

Constraints defined by different search tables are “OR-” combined. This means for an entry stored on the server, it must at least match one of the defined search tables to fit the

search: ?

☒ general information
 ☐ time model aspects
 ☒ data aspects
 ☐ representational Aspects
 ☐ supported tasks and application areas
 ☐ classification

☒ general information
 

☒ name
 ☐ references
 ☒ description
 ☐ url

☐ data aspects
 

☐ data types
 

☐ nominal
 ☐ ordinal
 ☐ discrete
 ☐ continuous
 ☐ binary

☐ structure
 

☐ univariate
 ☐ map
 ☐ 3- dimensional
 ☐ multivariate
 ☐ hierarchical
 ☐ network

add Search Table

specify query: (↔: and; ↑: or)

close

general information	
name	description

search

show all

☐ update search results automatically

Figure 41: The “search area” with one “search table”

search request.

Constraints defined by the different columns in a search table are “AND-” combined. So for a stored entry to fit the search constraints defined by a search table, its values must match the search values defined in the search table (the rules for when these values “match” are defined below).

Intentionally, a search table is an “example” table the user can use to define which values the (to him/her) relevant part of the view table shall contain.

Following rules are applied to check whether a search value matches the corresponding value of a stored entry: Columns/attributes that are not part of a search table are neglected. So the search result is independent of the values stored in the database for these attributes .

Otherwise, comparison of a value stored on the server with the search value depends on its data type:

- string, md5, text: For these data types, a fulltext search is performed. A match is given, if the search string appears somewhere in the string stored on the server. Thereby the “%” character can be used as a wildcard (that matches any string; it is not required at the beginning or end of a string). The search is case sensitive.
- lists (stringlist, urllist): For lists, a match is given if each specified list item matches at least one stored list item. For the list items, a fulltext search (see the previous rule) is performed. This means, the list stored on the server must at least contain

the elements defined by the search to fit the search.

- enum, fpn: These two types match, if the search value and the stored value are equal.
- set: A set type matches a search, if the set defined in the search is a subset (or the equal set) of the stored set. So in the set stored in the database, at least the values defined by the search must be selected.
- int: For the int type, two different search types are available: If a single number is defined, it is checked for equality with the stored value.

Additionally to this search for identical numbers, it is possible to search for number intervals. The interval can be defined by a lower and an upper bound. Each of the boundaries must be either an integer number or can be missing. Then the interval is unbound in this direction. Both boundaries must be separated by a “;” (semi-colon). This character must not be missing, even if one of the boundaries is not defined. The first and the last character of such an interval must be “[“ or “]”. For example [-1;1] means that both boundaries are included within the interval, whereas ]-1;1[ means that both boundaries are not part of the interval.

Therefore valid interval examples are ]-2;0], [;199[ or even ];[. Invalid intervals are [100] or ]100; .

(Insertion of search table values is identically to insertion of values to edit tables, and is therefore described in “Interfaces for the Different Data Types” in the section “Edit Page”).

If the auto-search functionality is disabled, the defined search request is only sent to the server when the “search” button is pressed, placed at the bottom of the search area. This button is only enabled, when at least one search table exists (Fig. 41).

To display all entries stored in the database, the “show all” button must be pressed (Fig. 41).

## Creating Search Tables

To create a new search table, in a first step the desired columns must be specified. This is done using the elements at the top of the search area (labelled with “A” in Fig. 41). With two exceptions they are working the same way as the control checkboxes for the visibility of the columns in the view table (see “Hide/Show Columns”). First, the tables storing the checkboxes for the search tables are faded out (invisible) by default. And whenever one of these tables is faded out, all checkboxes in this table become unchecked.

All columns whose corresponding checkbox is checked are selected. If at least one column is selected, the “add Search Table” button is enabled, otherwise it’s disabled. Clicking onto this button creates a new search table that consists of the defined columns.

To remove a search table, the button labelled with “close” in the left upper corner of the search table must be clicked (Fig. 41).

## Auto - Search

DQT provides the possibility to send the search request automatically to the server (and therefore automatically updates the “view table”), whenever any value in the search tables (or the number of search tables) has changed. To activate this functionality, one has to check the checkbox beneath “search” and “show all” buttons. With this function

enabled, the search request is not sent immediately to the server, but only after a timer of 3 seconds. If another change on the search request is detected within these 3 seconds, the countdown is resetted and the time is started again.

### Performing a search

Once a search request has been sent to the server (e.g. explicitly using the “search” button, or by the auto- search function), an acknowledge message is shown. The loading message in the view area is displayed too, to indicate that new data is loaded for the view table. All rows that do not match the search constraints are removed from the view table as well as from the hidden table. If no row matches the search values, the view table is replaced by the message “no entry found”.

Rows that were not in the result set of the last search, but now are, are inserted at the correct position in the view table to keep it sorted. Rows that fell out of the result set while they have been hidden are reinserted into the hidden table when they become part of the result set again. They need to be explicitly inserted to the view table by the user. The number of rows found is displayed at the bottom of the search area (Fig. 42). This information is removed when the user changes the definition of the search request.



Figure 42: The “search” and “show all” button, the checkbox that controls the auto-search function, and the number of entries that fit the last search

### 10.4.6 Export Area

The third part of the view page is the export area (Fig. 43).



Figure 43: The export area

It consists of the following elements:

- A selection box, which contains a list of available output formats.
- An “export” button, that starts the export process.
- An “update export results automatically” checkbox, that controls the auto export functionality.
- A text area, where the result of the output is displayed.

The export area is closed by default.

### 10.4.7 Export

DQT provides the possibility to export the structure and the content of the view table to different data formats.

To export the current view table (that means only the rows that are currently shown in the view table, and only the visible columns), the output format must be chosen in the selection box first. Then the “export” button must be pressed. This sends the export request to the server. The results returned from the server are written into the text area.

#### Auto - Export

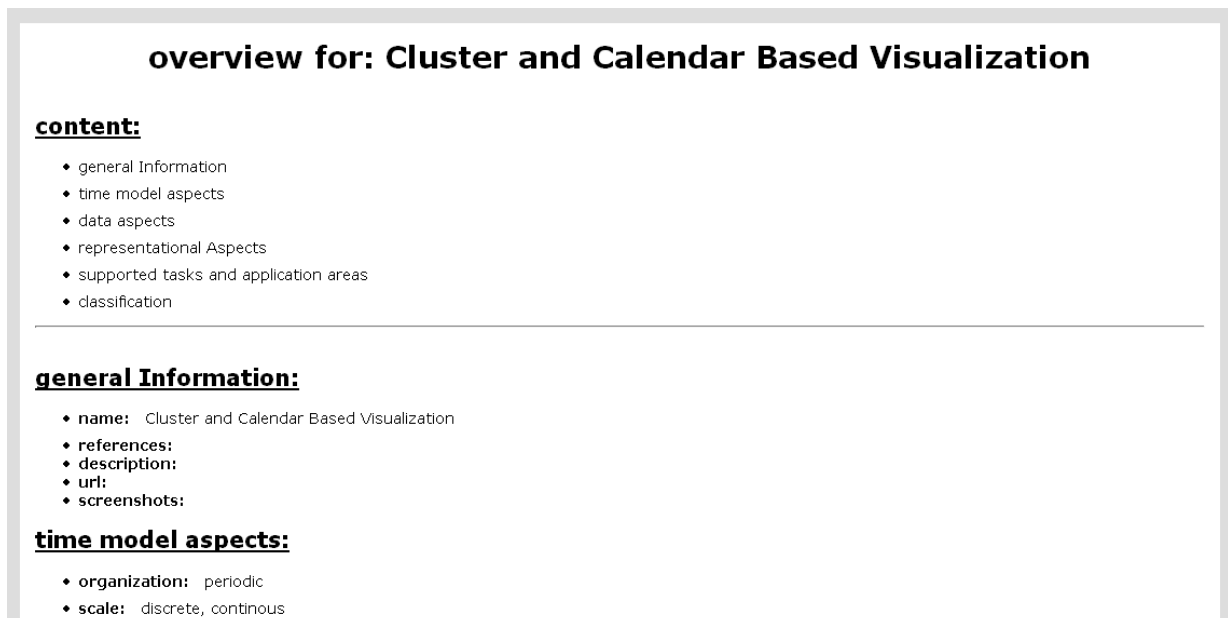
DQT allows an automatic update of the exported data, whenever a change is made to the view table. Thereby “making a change” means to show/hide a row or column, to receive the result of a search or to choose another export format.

To activate this function, the “update export result automatically” checkbox must be checked. If this function is active, each change on the page that makes the result of the last export inconsistent with the current view table starts a timer of 3 seconds. After these 3 seconds, the current “view table” is automatically exported. If another change to the view table is made within these 3 seconds, the countdown is resetted and the timer is started again.

## 10.5 Detail-/Entry Window

The only content of this window is the detail- or entry page.

It presents the information of exactly one entry of DQT (That’s why this page is called detail- or entry page: detail because it shows detailed information about one entry, or entry because information about exactly one entry is shown on the page).



The screenshot shows a web page titled "overview for: Cluster and Calendar Based Visualization". The page is structured with several sections, each starting with a bold header and followed by a bulleted list of details. The sections are: "content:" (with 6 items), "general Information:" (with 5 items), and "time model aspects:" (with 2 items). The page has a clean, minimalist design with a light gray background and a white content area.

**overview for: Cluster and Calendar Based Visualization**

**content:**

- general Information
- time model aspects
- data aspects
- representational Aspects
- supported tasks and application areas
- classification

---

**general Information:**

- **name:** Cluster and Calendar Based Visualization
- **references:**
- **description:**
- **url:**
- **screenshots:**

**time model aspects:**

- **organization:** periodic
- **scale:** discrete, continuous

Figure 44: Example for a detail- or entry page

The information about the entry is structured according to the attribute classification. At the top of the page there’s a simple navigation that allows to jump quickly to different

topics, into which the information is separated. Its main functionality is to present the different entries stored in the DQT.

This page can be accessed by clicking onto the links in the first cell of the rows of the “view table” or the tables on the edit page. The page can also be accessed or linked directly using the URL “<applRoot>/entryPage.php?dataId={idOfEntry}”. Thereby <applRoot> is the base URL where the instance of DQT is placed, and {idOfEntry} is a unique integer value DQT automatically assigns to each entry. If the default user (a user that is not logged in) does not have the “read” permission, only an error message is displayed.



# 11 Documentation of the Implementation

## 11.1 Application Structure

This section describes the components of DQT and how they are connected.

### 11.1.1 Components of DQT

The several components of DQT are visualized in the UML component diagram shown in Figure 45.

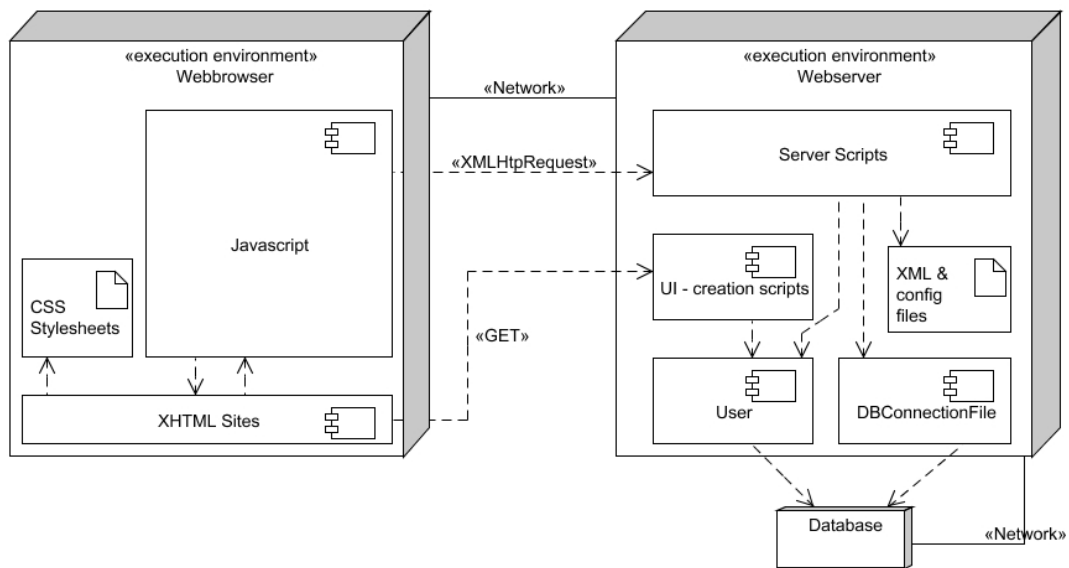


Figure 45: UML component diagram of DQT

At most installations of DQT, the database is probably physically located on the same server machine as the webserver. Currently, DQT only runs with a MySQL database. The “UI - creation scripts” are those php-files, which are used to create the XHTML pages that are returned in response to a clients GET request for those files. The “Server Scripts” component includes all php files that implement functionality called asynchrony by the JavaScript component (e.g. save, delete, load) implementing the application logic on the client as well as the asynchronous communication with the server. All files belonging to the “Server Script” component access the database using the “DB-ConnectionFile” component, whereas the “User” component (responsible to allow or deny access to functionalities of DQT, as well as to manage the login/logout of users) has its own database connection.

### 11.1.2 Overview over the Configuration Files

This section describes the purpose of the files belonging to the “XML & config files” component.

`<applRoot>/conf/config.php`: The main configuration file. For more information about this file please see section 9.8 “Configuration File `<applRoot>/conf/config.php`”. This file must have exactly this name.

`<applRoot>/conf/config.xml`: Contains an absolute path to the root directory of the application that can be accessed during XSLT transformation. This file must have exactly this name.

`<applRoot>/XMLData/tableStructure.xml`: Contains the definition of the table structure (which columns and how are they grouped). See section 9.2.2 “Table Structure Definition File”. The name of this file can be chosen arbitrarily as it must be defined in the `<applRoot>/conf/config.php` file.

`<applRoot>/XMLData/tableStructure_users.xml`: Contains the definition of the structure of the table where the users, their passwords and rights are stored. See section 9.10.3 “Management and Storage of User Data”. As for `tableStructure.xml`, its name can be chosen arbitrarily but must be defined in `<applRoot>/conf/config.php`.

`<applRoot>/XMLData/TS2editTS.xsl`: An XSLT stylesheet that defines the elements from `tableStructure.xml` to be shown on the edit page. See 9.4 “Defining the Attributes to be Displayed” for details. The name of this file can be chosen arbitrarily and must be defined in `conf/config.php`.

`<applRoot>/XMLData/TS2viewTS.xsl`: An XSLT stylesheet that defines the elements from `tableStructure.xml` to be shown on the view page. See 9.4 “Defining the Attributes to be Displayed” for details. The name of this file can be chosen arbitrarily and must be defined in `conf/config.php`.

`<applRoot>/XMLData/edittableHeader.xsl`: An XSLT stylesheet that defines the transformation from the content of `tableStructure.xml` to the tables displayed on the edit page.

`<applRoot>/XMLData/viewTableHeader.xsl`: An XSLT stylesheet that defines the transformation from the content of `tableStructure.xml` to the table displayed on the view page.

`<applRoot>/XMLData/exportStylesheets`: This folder contains the XSLT stylesheets that can be used to transform the output of DQT. See 9.7.3 “Export Stylesheets” for more information. This folder must have this name.

`<applRoot>/XMLData/attributeTypeBinding.xsl`: Creates pairs of `[attributeId]: [type-String]`; out of an table structure XML file. Must have exactly this name.

`<applRoot>/XMLData/dataSite.xsl`: The stylesheet that is used to create the detail page out of the XML formatted output of the export function. Must have exactly this name.

`<applRoot>/XMLData/emptyNode.xml`: Only contains an empty XML node that is needed for some XSLT transformations. Must have exactly this name.

`<applRoot>sessionManagement.php`: This file is responsible for creating or updating the php-session on the webserver. It also checks whether a user object exists, and if not, it creates a user that is not logged in. The user object is stored in `$_SESSION["user"]`. The session id must be either submitted as value of a cookie, or, if the client does not allow cookies, as value of the URL parameter `PHPSESSID`.

This file must be included by all server scripts.

### 11.1.3 Server

In this section, we give an overview of the structure of the server side application logic of DQT, and about the dependencies between the files. Most of the server side application is not implemented using an object oriented approach, but an imperative one. Thereby, for

each action the client can request, a php file exists that implements a script which performs the necessary operations.

Because of the number of files and their relations, showing all server side files and relations in one huge diagram might only create chaos.

So the first (“Overview”) paragraph shows all existing files and gives a short description of their tasks. Then for each file its dependencies and relations to other files are other shown. Thereby files required for and used by each particularly file are visualized, and a list of files that require or use this file is provided.

## Overview

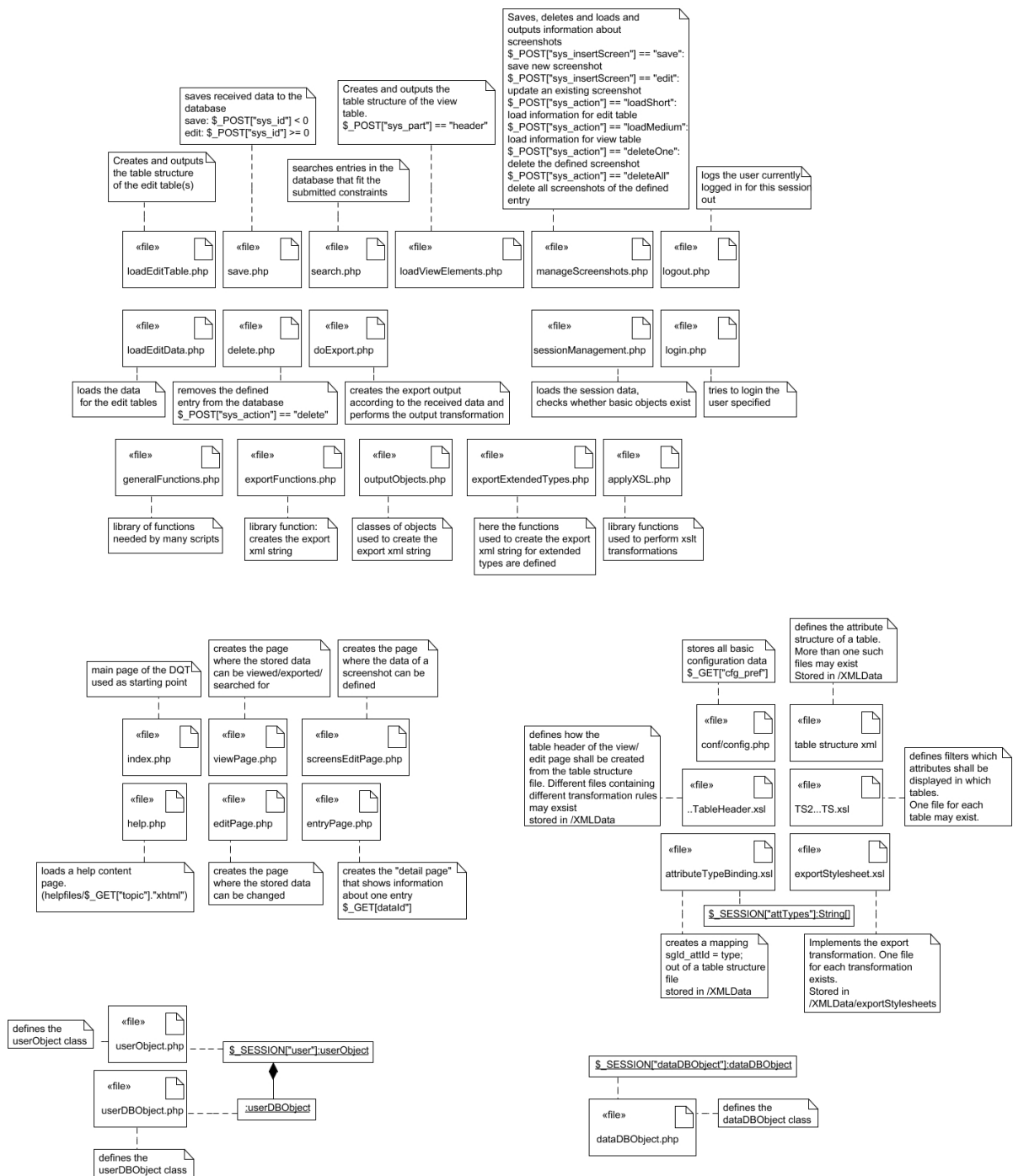


Figure 46: The files implementing the server side application logic, grouped according to the components shown in Figure 45

## applyXSL.php

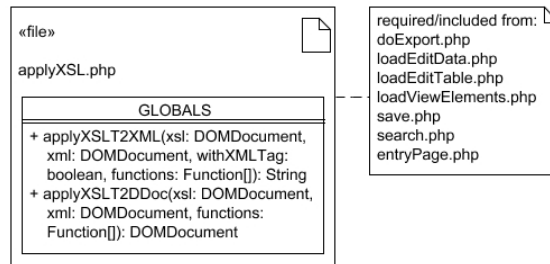


Figure 47: UML diagram showing the relations of `<applRoot>/applyXSL.php`

## dataDBObject.php

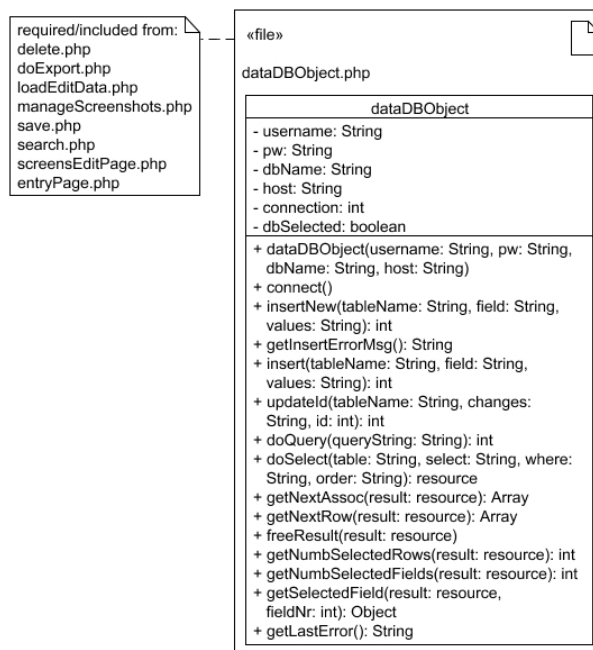


Figure 48: UML diagram showing the relations of `<applRoot>/dataDBObject.php`

## delete.php

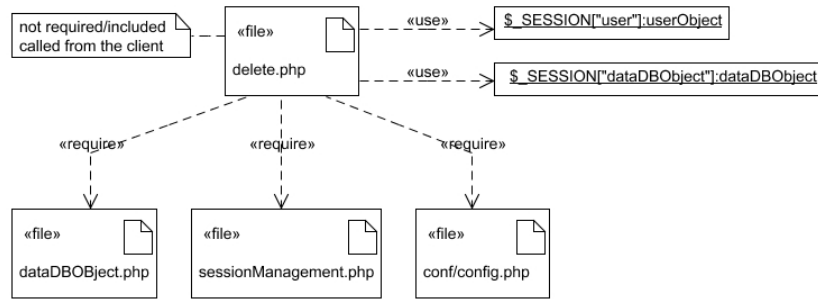


Figure 49: UML diagram showing the relations of `<applRoot>/delete.php`

## doExport.php

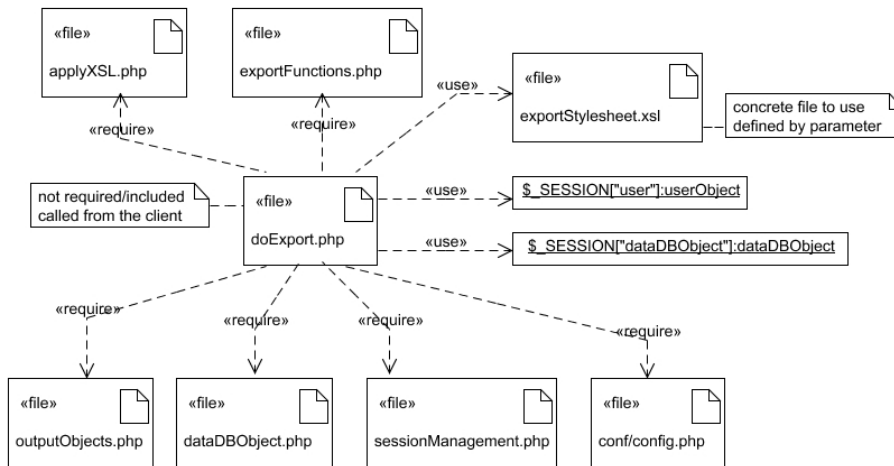


Figure 50: UML diagram showing the relations of `<applRoot>/doExport.php`

## editPage.php

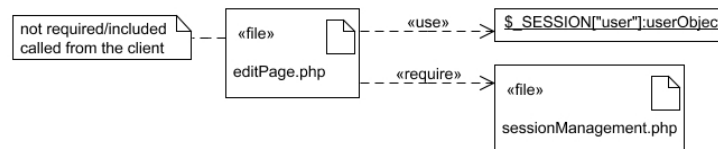


Figure 51: UML diagram showing the relations of `<applRoot>/editPage.php`

## entryPage.php

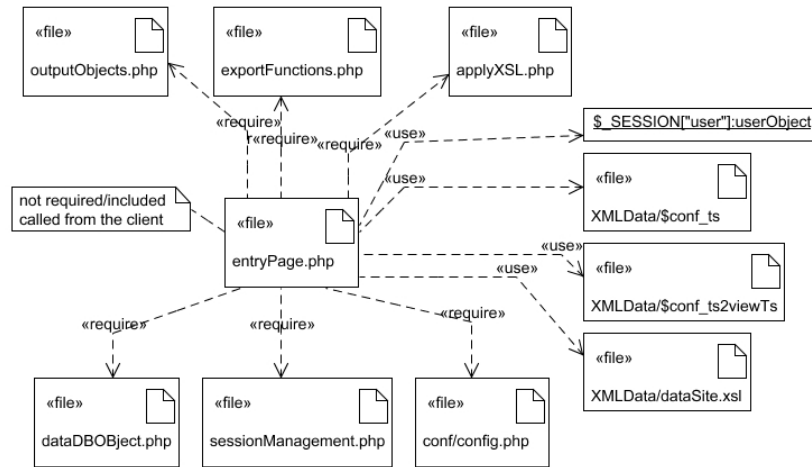


Figure 52: UML diagram showing the relations of `<applRoot>/entryPageet.php`

## exportExtendedTypes.php

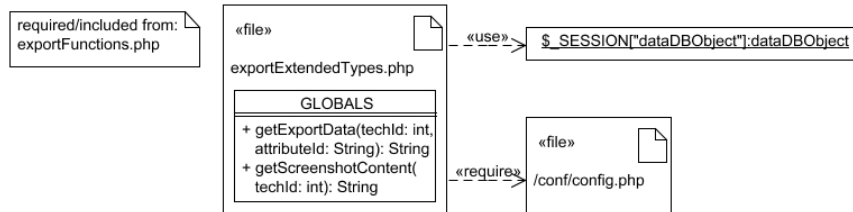


Figure 53: UML diagram showing the relations of `<applRoot>/exportExtendedTypes.php`

## exportFunctions.php

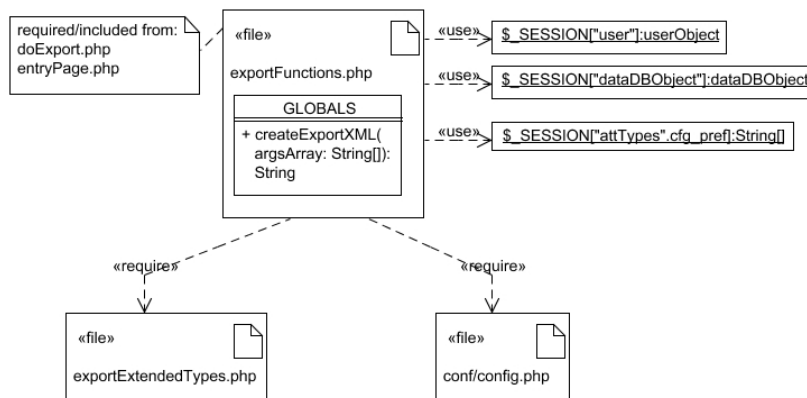


Figure 54: UML diagram showing the relations of `<applRoot>/exportFunctions.php`

## generalFunctions.php

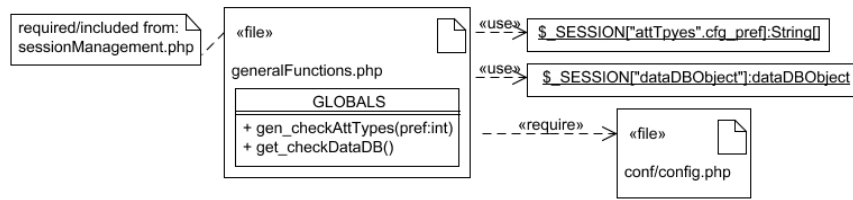


Figure 55: UML diagram showing the relations of `<applRoot>/generalFunctions.php`

## help.php

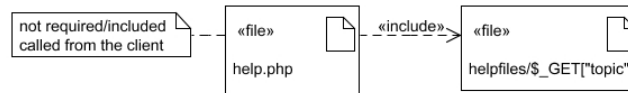


Figure 56: UML diagram showing the relations of `<applRoot>/help.php`

## index.php

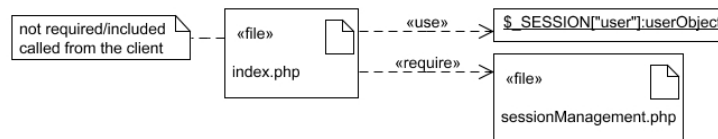


Figure 57: UML diagram showing the relations of `<applRoot>/index.php`

## loadEditData.php

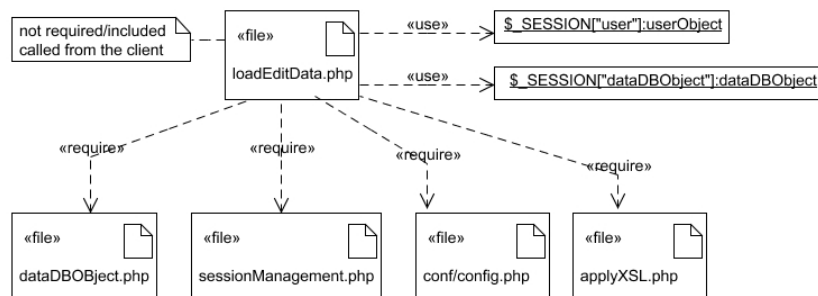


Figure 58: UML diagram showing the relations of `<applRoot>/loadEditData.php`



## loadEditTable.php

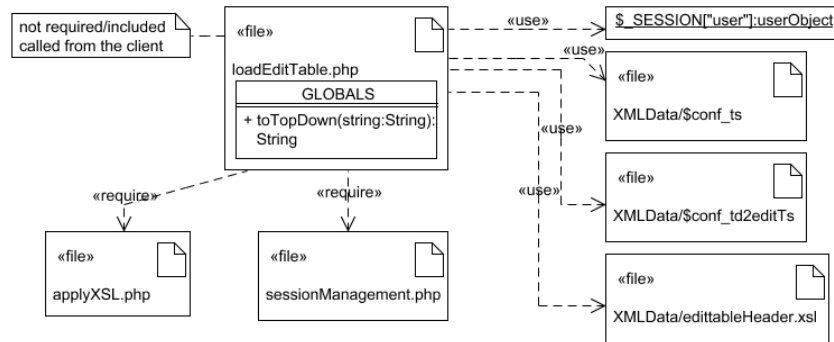


Figure 59: UML diagram showing the relations of `<applRoot>/loadEditTable.php`

## loadViewElements.php

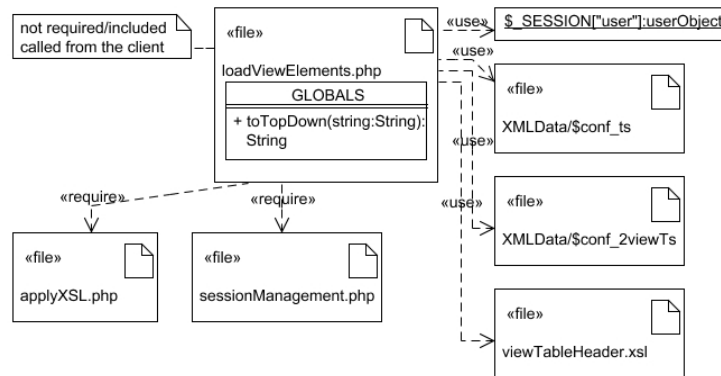


Figure 60: UML diagram showing the relations of `<applRoot>/loadViewElements.php`

## login.php

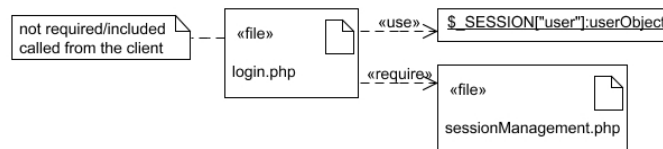


Figure 61: UML diagram showing the relations of `<applRoot>/login.php`

## logout.php

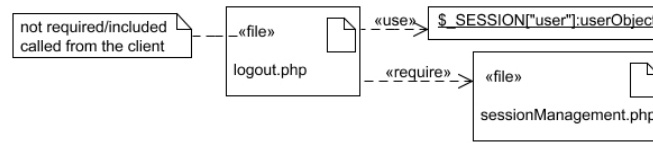


Figure 62: UML diagram showing the relations of `<applRoot>/logout.php`

## manageScreenshots.php

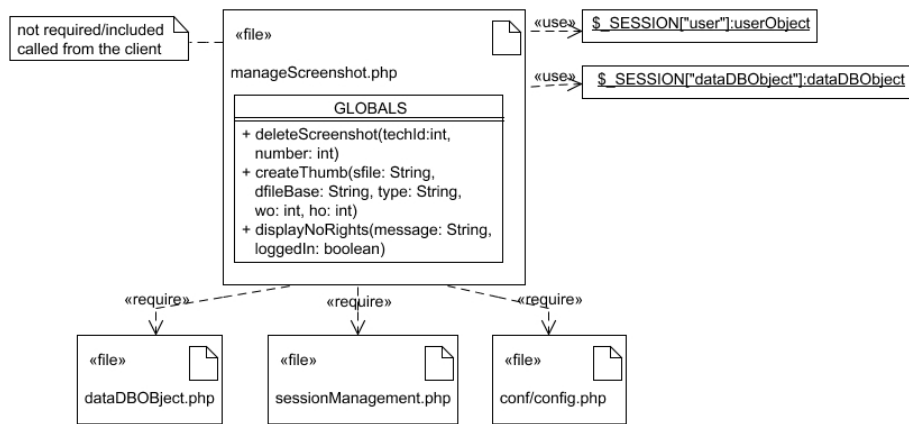


Figure 63: UML diagram showing the relations of `<applRoot>/manageScreenshots.php`

## outputObjects.php

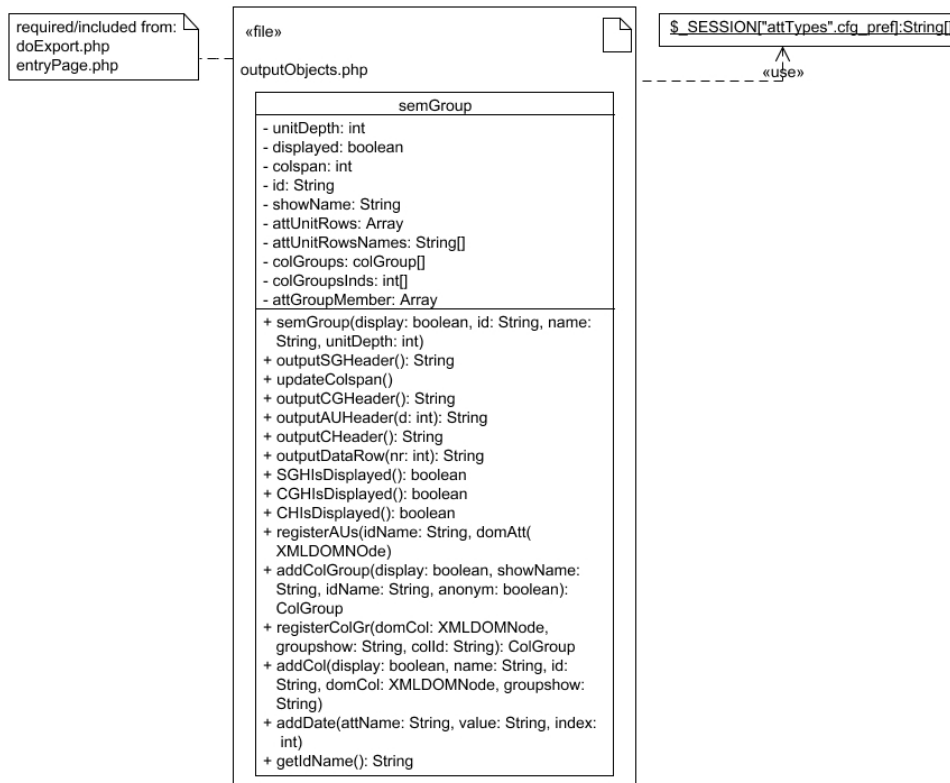


Figure 64: UML diagram showing the relations of `<applRoot>/outputObjects.php`

## save.php

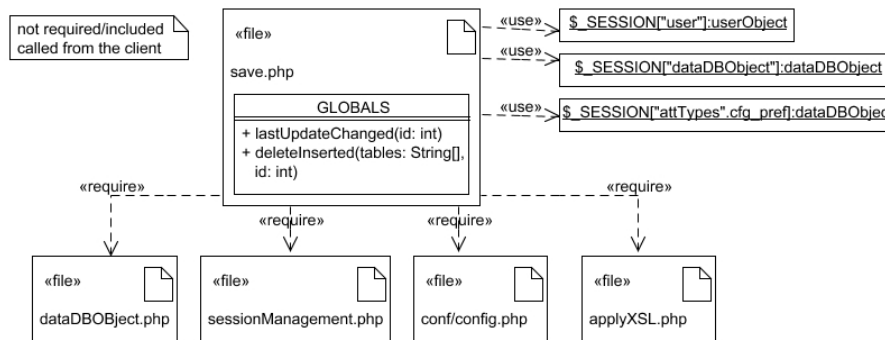


Figure 65: UML diagram showing the relations of `<applRoot>/save.php`

## screensEditPage.php

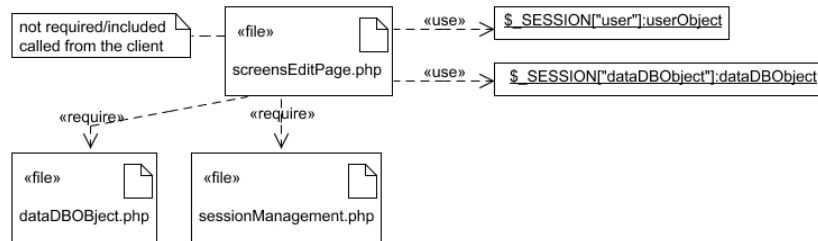


Figure 66: UML diagram showing the relations of `<applRoot>/screensEditPage.php`

## search.php

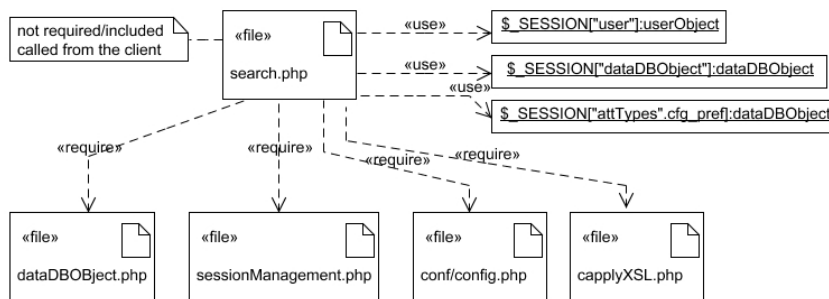


Figure 67: UML diagram showing the relations of `<applRoot>/search.php`

## sessionManagement.php

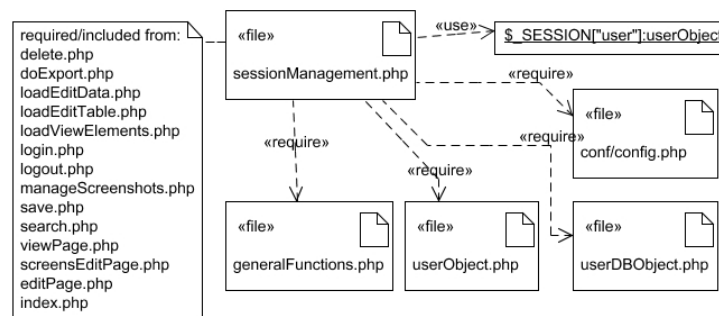


Figure 68: UML diagram showing the relations of `<applRoot>/sessionManagement.php`

## userDBObject.php

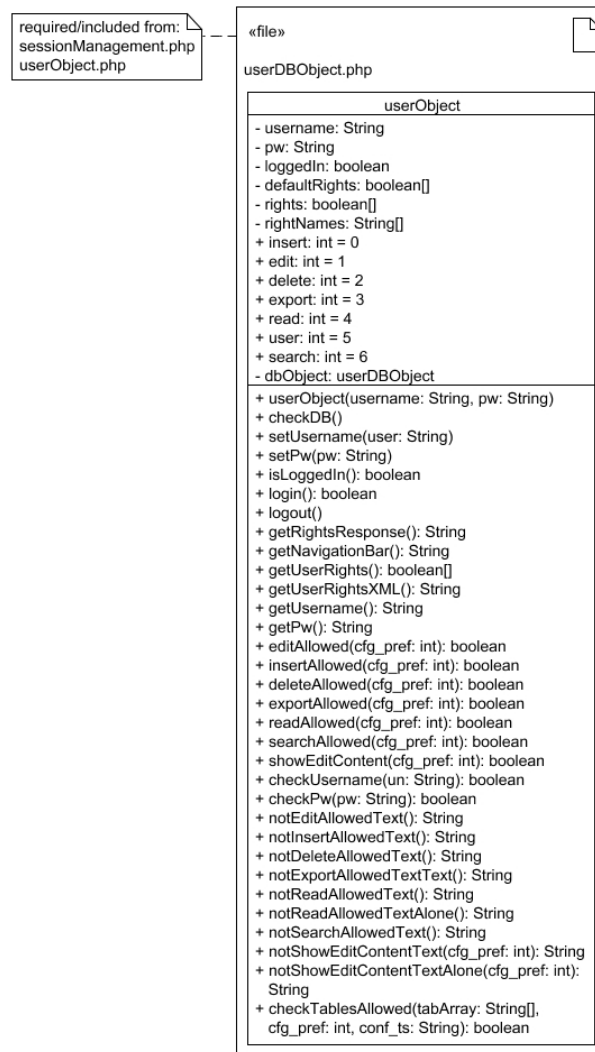


Figure 69: UML diagram showing the relations of *<applRoot>/userDBObject.php*

## userObject.php

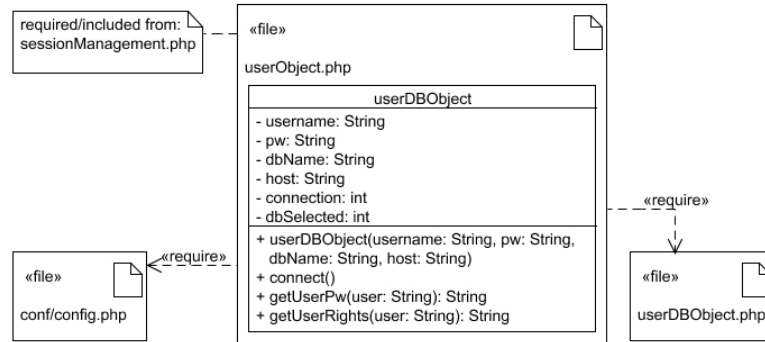


Figure 70: UML diagram showing the relations of `<applRoot>/userObject.php`

## viewPage.php

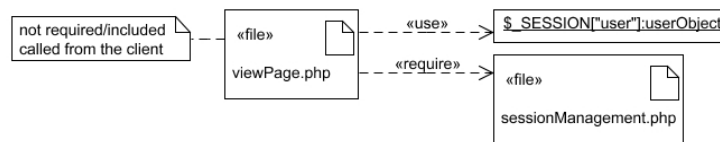


Figure 71: UML diagram showing the relations of `<applRoot>/viewPage.php`

### 11.1.4 Client

The client application logic is implemented using JavaScript. This section gives an overview about the structure of the JavaScript classes and their relations.

First, an overview over the existing packages and how they are related is given. Then the content of the packages is presented and how the classes within a package are related to each other. Thereby, all classes belonging to the same package are implemented in the same `.js` file, so each file represents one package.

At the end of this section it is shown which classes are used in the edit-/view part of the application.

### Packages

Figure 72 shows the existing JavaScript packages, which classes they contain and their relations. Thereby the hierarchy of the classes in the different table packages is most important. In the “searchHandlers” package, the classes “SearchQueryTableCluster”, “SearchQueryRow” and “SearchQueryCell” are subtypes of the according classes in the “editableTable” package, whereas the classes “SearchTableDefinerManager”, “SearchTableDefinerSGHandler”, “SearchTableDefinerCGHandler” and “SearchTableDefinerColumnHandler” inherit from the according classes in the “tableDisplayController” package.

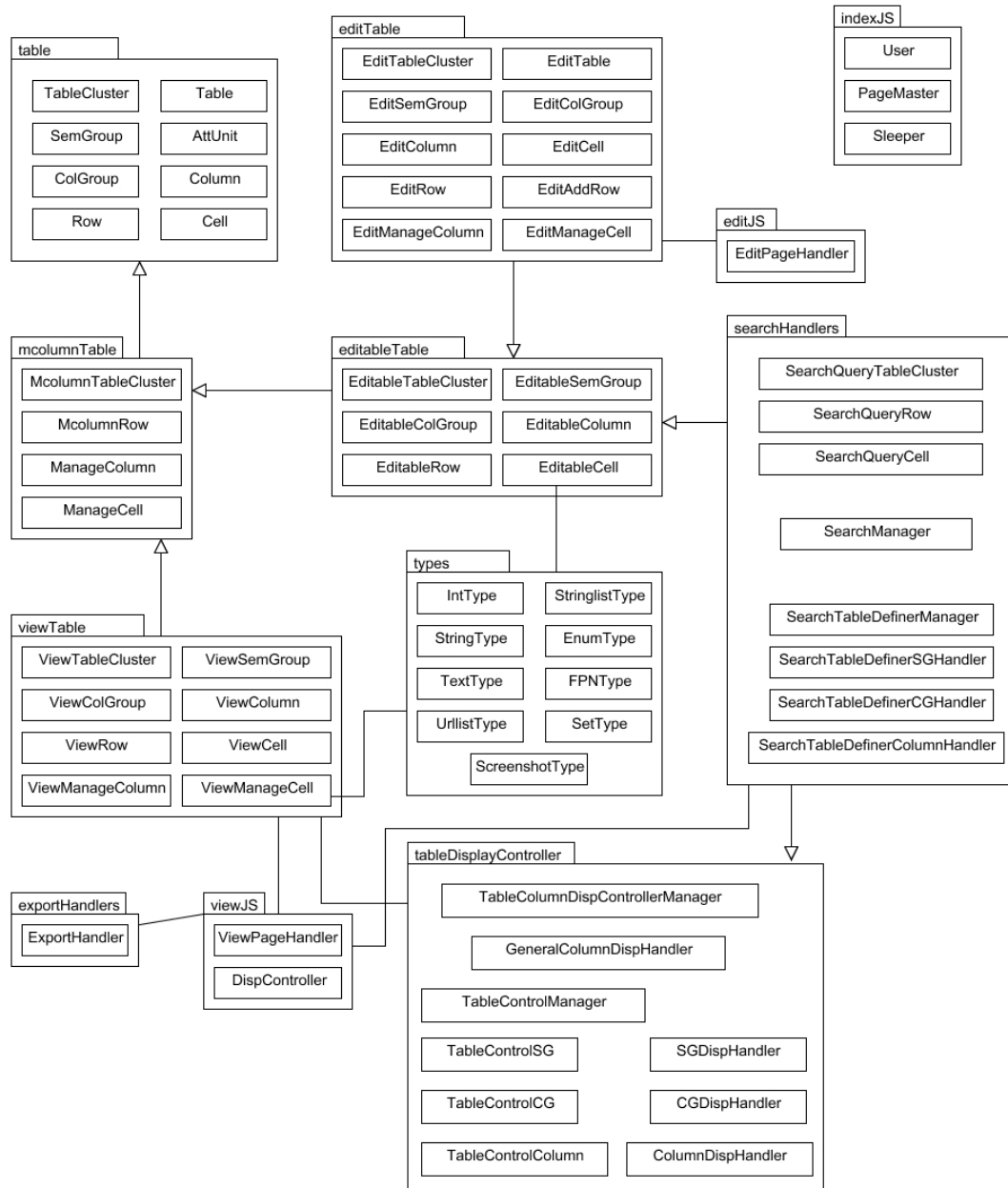


Figure 72: The content of the JavaScript packages and their relations

## indexJS.js

The classes in the `indexJS` package implement general functionalities, and are needed in both, the edit and the view part of DQT, and even on the index page.

User	PageMaster	GLOBALS
<div>- loginfile: String - logoutfile: String - mode: int - username: String - pw: String - loggedIn: boolean - rights: boolean[] - insert: int = 0 + edit: int = 1 + del: int = 2 + exp: int = 3 + read: int = 4 + users: int = 5 + search: int = 6 - loginXmlHttp: XMLHttpRequest - loginForm: HTMLDOMNode - loggingInText: HTMLDOMNode  + User(loginfile: String, logoutfile: String, rightsXMLS: String, mode: int) + parseServerResponse(xml: XMLDOMNode) + handleRights(actNode: XMLDOMNode) + serverAnswered() + login() + getUserData(): String + setLoggedIn(username: String, pw: String) + insertAllowed(): boolean + deleteAllowed(): boolean + readAllowed(): boolean + searchAllowed(): boolean + exportAllowed(): boolean + editAllowed(): boolean + logout()</div>	<div>- urlPrefix: String - sessionId: String - sessionXML: XMLHttpRequest - noTimeOutTimer: int - browserHasIEProblems: boolean  + PageMaster(urlPrefix: String, sessionId: String) + updateSession() + returnNull() + getXmlHttpRequest(): XMLHttpRequest + getXmlObjectFromString(xmlString: String): XMLDOMNode + sendXmlHttpRequest(data: String, retObj: Object, returnFunction: Function, method: String, url: String, aysnc: boolean, reqObject: XMLHttpRequest) + getRootNode(xml: XMLDOMNode) XMLDOMNode + getSessionId(): String + getUrlPrefix(): String + makeIERunnableClone(node:XMLDOMNode): HTMLDOMNode</div>	<div>+ userObj: User + pageMaster: PageMaster  + userRightsChanged() + getTextChildContent(node): String</div>
	<div>Sleeper</div> <div>- timeout: int - callData: Object  + Sleeper() + sleepViewLoad(sleepTime: int, data: Object) + wakeUpViewLoad() + sleepEditLoad(sleepTime: int, data: Object) + wakeUpEditLoad()</div>	

Figure 73: The content of the indexJS package

## table.js

The **table** package implements base classes for the general management of tables. These objects only implement general methods for the management and controlling. The objects in this package are completely independent of the environment they are used in (use no global variables,...).

This package implements the table model used by DQT (see 9.2.1 “Table Model”). It also supports the fact that a logical table can be split into different display tables, as it is done on the edit page of DQT.

Therefore **TableCluster** implements the huge logical table, whereas **Table** implements a displayed table that represents a part of a **TableCluster**.

For more information see the javadoc like documentation of the classes.



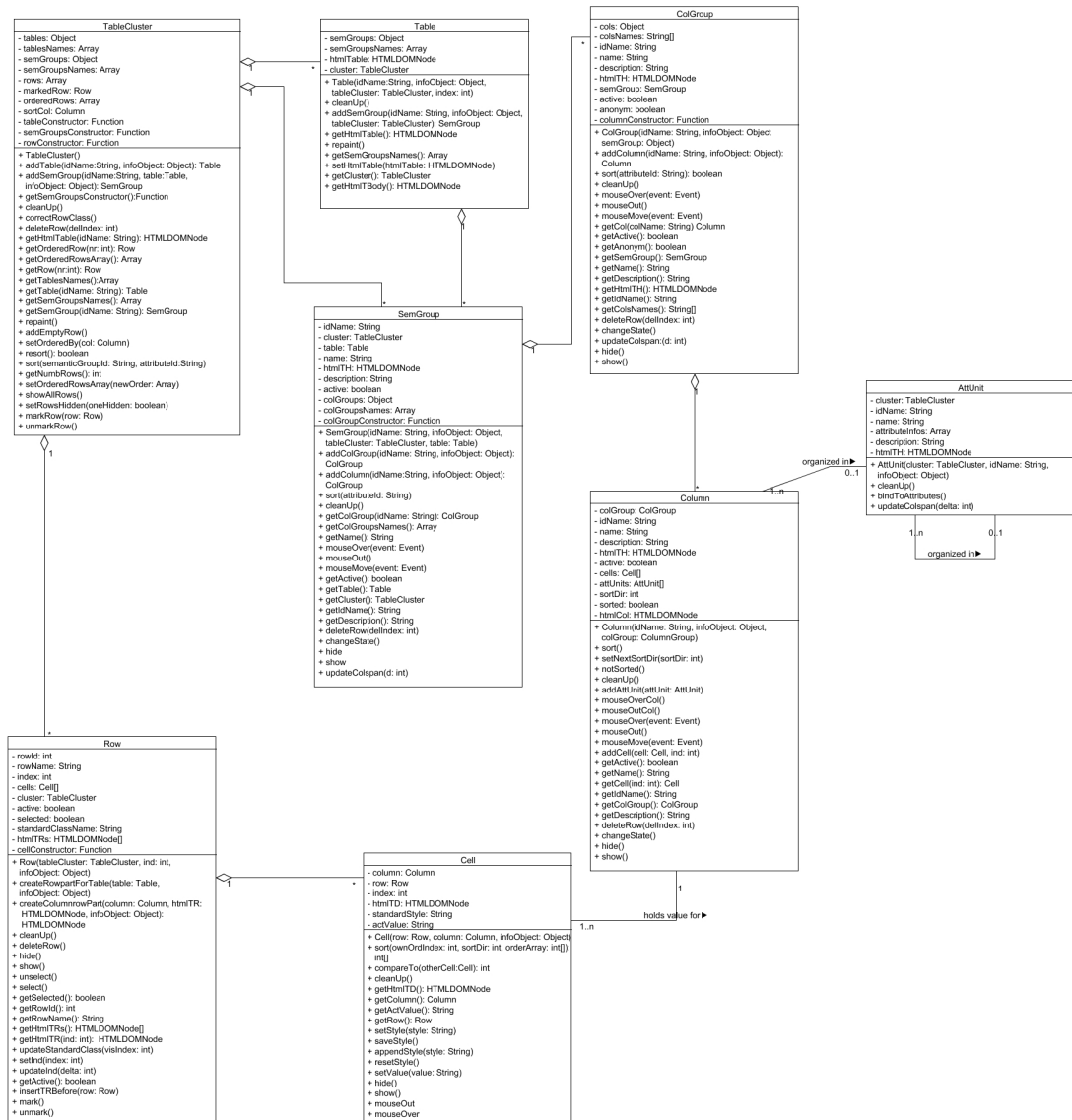


Figure 74: The content of the table package

## mcolumnTable.js

The `mcolumnTable` package extends the `table` package by adding a “management” (or “manage”) column. For each `Table` object, such a column exists. The idea of this column is to store for each row information that identify the row (like a name), and buttons/links that can be used to control the row.

For more information see the javadoc like documentation of the classes.

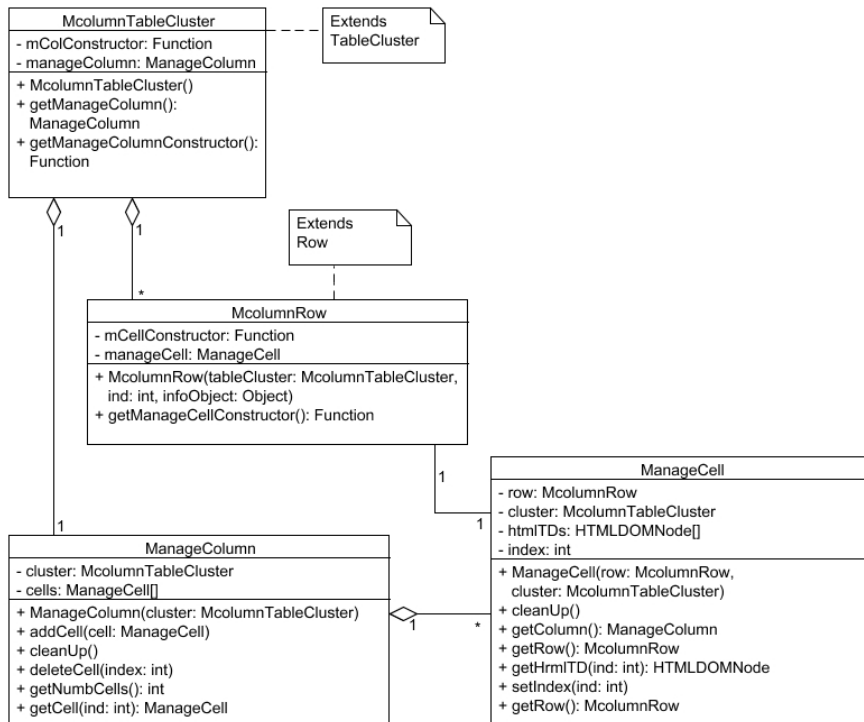


Figure 75: The content of the `mcolumnTable` package

## editableTable.js

The `editableTable` package extends the `mcolumnTable` package and is the direct base package for the packages that implement the edit tables and the search tables. It's main extension to the `mcolumnTable` package is, that the user interface of the cells now allow the user to enter a value.

For more information see the javadoc like documentation of the classes.

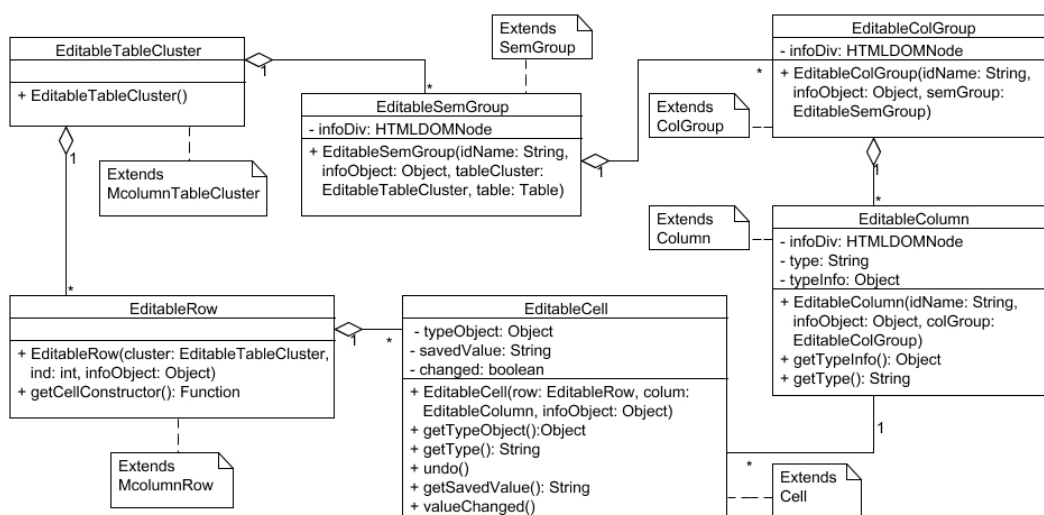


Figure 76: The content of the `editableTable` package

## types.js

The `type` package collects all implementations of type objects.

The global function `getExtendedType(idName, cellObj, edit)` returns a type object for the specified extended type.

For more information see the javadoc like documentation of the classes.

<b>IntType</b> - edit: boolean - cellObj: Cell - description: String - tooltip: String - allowIntervalValues: boolean - input: HTMLDOMNode + IntType(cellObj: Cell, edit: boolean) + enableIntervalValues() + doFocus() + changeValue() + setValue(value: String) + showValue(valueId: String) + focused() + compare(myCell: Cell, otherCell: Cell): int	<b>TextType</b> - edit: boolean - cellObj: Object - description: String - tooltip: String - area: HTMLDOMNode - orderType: int + TextType(cellObj: Cell, edit: boolean) + setOrderType(type: int) + changeValue() + doFocus() + focused() + setValue(value: String) + showValue(valueId: String) + compare(myCell: Cell, otherCell: Cell): int	<b>SetType</b> - edit: boolean - cellObj: Object - tooltip: String - valueInfos: Array - selectedTableHtml: HTMLDOMNode - selectionBoxHtml: HTMLDOMNode - htmlUL: HTMLDOMNode + TextType(cellObj: Cell, values: XMLDOMNode, edit: boolean) + changeValue() + createSelectEntry(text: String, entryNr: int): HTMLDOMNode + addEntry() + selectEntry(entryNr: int) + unselectThisEntry() + unselectEntry(entryNr: int) + setValue(value: String) + showValue(valueId: String) + compare(myCell: Cell, otherCell: Cell): int
<b>StringType</b> - edit: boolean - cellObj: Object - description: String - tooltip: String - input: HTMLDOMNode + StringType(cellObj: Cell, edit: boolean) + doFocus() + focused() + changeValue() + setValue(value: String) + showValue(valueId: String) + compare(myCell: Cell, otherCell: Cell): int	<b>StringlistType</b> - edit: boolean - cellObj: Object - htmlUL: HTMLDOMNode - standardClass: String + UrlistType(cellObj: Cell, edit: boolean) + setValue() + showValue(valueId: String) + compare(myCell: Cell, otherCell: Cell): int	<b>ScreenshotType</b> - edit: boolean - cellObj: Object - specialSave: true - xmlHttpObj: XMLHttpRequest - htmlDiv: HTMLDOMNode - htmlSelect: HTMLDOMNode - htmlImg: HTMLDOMNode - imgDescriptionHtmlDiv: HTMLDOMNode - popup: Window - pics: Array + ScreenshotType(cellObj: Cell, values: XMLDOMNode, edit: boolean) + loadValues() + handleLoadedData() + handleData() + valueChanged() + save() + setValue(value: int) + getNumEntries(): int + wantDelete() + deleteEntry(id: int) + deletedOne() + oneDeleted(delId: int) + deletedAll() + wantAdd() + wantEdit() + update() + compare(myCell: Cell, otherCell: Cell): int
<b>EnumType</b> - edit: boolean - cellObj: Object - tooltip: String - disp: HTMLDOMNode - valueInfos: Object + TextType(cellObj: Cell, values: XMLDOMNode, edit: boolean) + changeValue() + focused() + setValue(value: String) + showValue(valueId: String) + getOrderValue(valueId: String): int + compare(myCell: Cell, otherCell: Cell): int	<b>UrlistType</b> - edit: boolean - cellObj: Object - htmlUL: HTMLDOMNode - standardClass: String + UrlistType(cellObj: Cell, edit: boolean) + setValue() + showValue(valueId: String) + compare(myCell: Cell, otherCell: Cell): int	
<b>GLOBALS</b> + getExtendedType(idName: String, cellObj: Cell, edit: boolean): Object	<b>FPNTType</b> - edit: boolean - cellObj: Cell - description: String - values: Array = {'f', 'p', 'n'} - tooltips: String[] - htmlImg = HTMLDOMNode - imgSrcs = String[] + FPNTType(cellObj: Cell, edit: boolean) + getValueIndex(value: char): int + changeValue() + setValue(value: char) + showValue(valueId: int) + compare(myCell: Cell, otherCell: Cell): int	

Figure 77: The content of the `types` package

## editJS.js

The `editJS` package contains all objects and functions necessary to manage the edit page. Especially this means to load and create the tables and to check whether the user has all necessary rights. The source code in this package is strongly dependent from the HTML page created by `<applRoot>/edit.php`, and needs to be changed when it shall be used elsewhere.

For more information see the javadoc like documentation of the classes and functions.

EditPageHandler	GLOBALS
<ul style="list-style-type: none"> <li>- editTableObject: boolean</li> <li>- loadfile: fileUrl</li> <li>- loadXmlHttp: XMLHttpRequest</li> <li>- saveUrl: String</li> <li>- loadDataUrl: String</li> <li>- deleteUrl: String</li> <li>- sleeper: Sleeper</li> <li>- sortSGId: String</li> <li>- sortAttrId: String</li> <li>- showDetailsLink: boolean</li> </ul>	<ul style="list-style-type: none"> <li>+ editTable: EditTableCluster</li> <li>+ userRightsChanged(loggedIn: boolean)</li> </ul>
<ul style="list-style-type: none"> <li>+ EditPageHandler(fileUrl: String, saveUrl: String, getDataUrl: String, deleteUrl: String, sortSGId: String, sortAttrId: String, showDetailsLink: boolean)</li> <li>+ load()</li> <li>+ receiveTableHeader()</li> <li>+ receiveTableData()</li> <li>+ parseTableHeader(xml: XMLDOMNode): boolean</li> <li>+ parseTableData(xml: XMLDOMNode)</li> <li>+ parseRows(data: object)</li> <li>+ updateLoadingStatus(text: String)</li> <li>+ handleTableInfos(htmlTable: HTMLDOMNode)</li> <li>+ handleSemGroupInfo(into: XMLDOMNode, htmlTable: HTMLDOMNode, htmlTH: HTMLDOMNode): EditSemGroup</li> <li>+ handleAttGroupInfo(into: XMLDOMNode, htmlTH: HTMLDOMNode): EditTabColGroup</li> <li>+ handleAttributeInfo(into: XMLDOMNode, htmlTH: HTMLDOMNode): EditTabCol</li> <li>+ getSaveUrl(): String</li> <li>+ getDeleteUrl(): String</li> <li>+ getTableObject(): EditTableObject</li> <li>+ removeTableObject()</li> </ul>	

Figure 78: The content of the `editJS` package

## `editTable.js`

The `editTable` package extends the table implemented in the `editable` package. It implements the table objects used on the edit page. The functionality provided by this package includes saving or deleting of entries.

For more information see the javadoc like documentation of the classes.

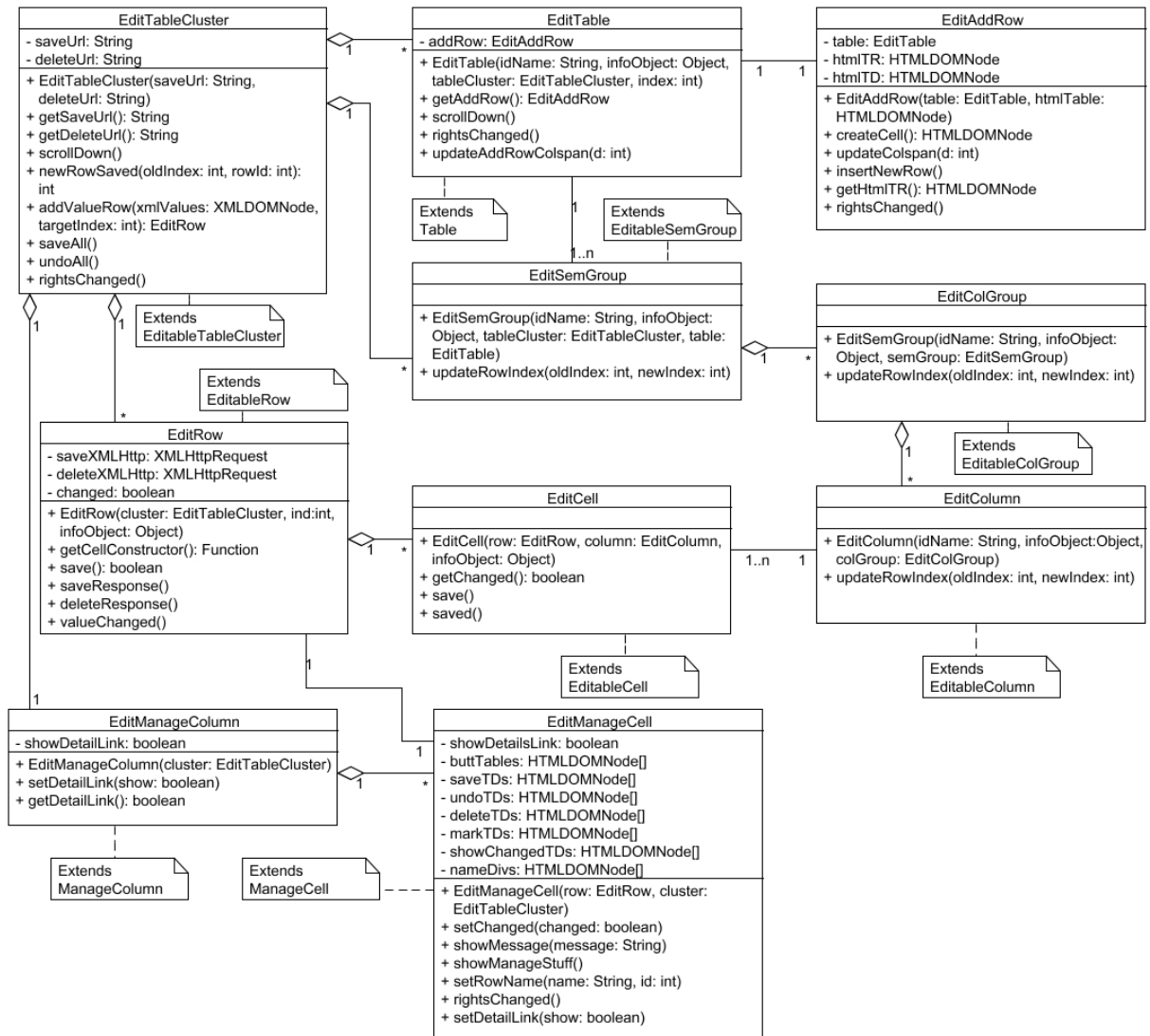


Figure 79: The content of the editTable package

## viewJS.js

The **viewJS** package contains all objects and functions necessary to manage the view page. This includes the loading and creating of the view table. It also contains the code that expands/collapses the different areas of the view page and controls which functionalities the user is allowed to use. The sourcecode in this package is strongly dependent from the HTML page created by `<applRoot>/edit.php`, and needs to be changed when it shall be used elsewhere.

For more information see the javadoc like documentation of the classes and functions.

DispController	ViewPageHandler
<ul style="list-style-type: none"> <li>- seeSearch: boolean</li> <li>- seeView: boolean</li> <li>- seeChooseView: boolean</li> <li>- seeViewTable: boolean</li> <li>- seeHiddenRows: boolean</li> <li>- seeExport: boolean</li> <li>- attached: int</li> <li>- chooseViewDiv: HTMLDOMNode</li> <li>- chooseViewImg: HTMLDOMNode</li> <li>- viewTableDiv: HTMLDOMNode</li> <li>- viewTableImg: HTMLDOMNode</li> <li>- viewHiddenRowsDiv: HTMLDOMNode</li> <li>- viewHiddenRowsImg: HTMLDOMNode</li> <li>- seeImg: HTMLDOMNode</li> <li>- seeDiv: HTMLDOMNode</li> <li>- exportImg: HTMLDOMNode</li> <li>- exportDiv: HTMLDOMNode</li> <li>- saveBlock: HTMLDOMNode</li> <li>- tellBlock: HTMLDOMNode</li> </ul>	<ul style="list-style-type: none"> <li>- loadfile: String</li> <li>- searchUrl: String</li> <li>- exportUrl: String</li> <li>- loadXmlHttp: XMLHttpRequest</li> <li>- attUnitArray: Array</li> <li>- nothingToShowElement: HTMLDOMNode</li> <li>- exportObject: ExportHandler</li> <li>- readTableDisplayHandler: TableColumnDispControllerManager</li> <li>- searchTableDefiner: SearchTableDefinerManager</li> <li>- sleeper: Sleeper</li> </ul>
<ul style="list-style-type: none"> <li>+ DispController()</li> <li>+ changeDisplay(dsp: String)</li> <li>+ setDisplay(dsp: String, see: boolean)</li> <li>+ hideForLoading()</li> <li>+ showAfterLoading()</li> </ul>	<ul style="list-style-type: none"> <li>+ ViewPageHandler(fileUrl: String, searchUrl: String, exportUrl: String)</li> <li>+ deleteSearchTableDefiner():</li> <li>+ getXmlHttp():XMLHttpRequest</li> <li>+ load()</li> <li>+ receiveTableHeader()</li> <li>+ receiveTableData()</li> <li>+ finishedDataReceiving()</li> <li>+ showLoadMessage()</li> <li>+ hideLoadMessage()</li> <li>+ parseTableHeader(xml: XMLDOMNode): boolean</li> <li>+ parseTable(xml: XMLDOMNode)</li> <li>+ parseRows(data: Object)</li> <li>+ updateLoadingStatus(text:String)</li> <li>+ handleTableInfos(htmlTable: HTMLDOMNode, vTN: boolean, sTN: boolean, xmlTable: XMLDOMNode)</li> <li>+ handleSemGroupInfo(info: XMLDOMNode, htmlTable: HTMLDOMNode, htmlTH: HTMLDOMNode, vTN: boolean, sTN: boolean): ViewSemGroup</li> <li>+ handleAttGroupInfo(info: XMLDOMNode, htmlTH: HTMLDOMNode, vTN: boolean, sTN: boolean): ViewColGroup</li> <li>+ handleAttUnitInfo(info: XMLDOMNode, htmlTH: HTMLDOMNode): ViewAttUnit</li> <li>+ handleAttributeInfo(info: XMLDOMNode, htmlTH: HTMLDOMNode, vTN: boolean, sTN: boolean): ViewTabCol</li> </ul>

Figure 80: The content of the **viewJS** package

## viewTable.js

The **viewTable** package extends the **mcolumnTable** package. It implements the objects used to create and manage the view table on the view page. The additional functionalities include the handling of hidden rows in the hidden table and the possibility to register an event listener that is called whenever content of the view table or the number of its columns or rows is changed.

For more information see the javadoc like documentation of the classes.

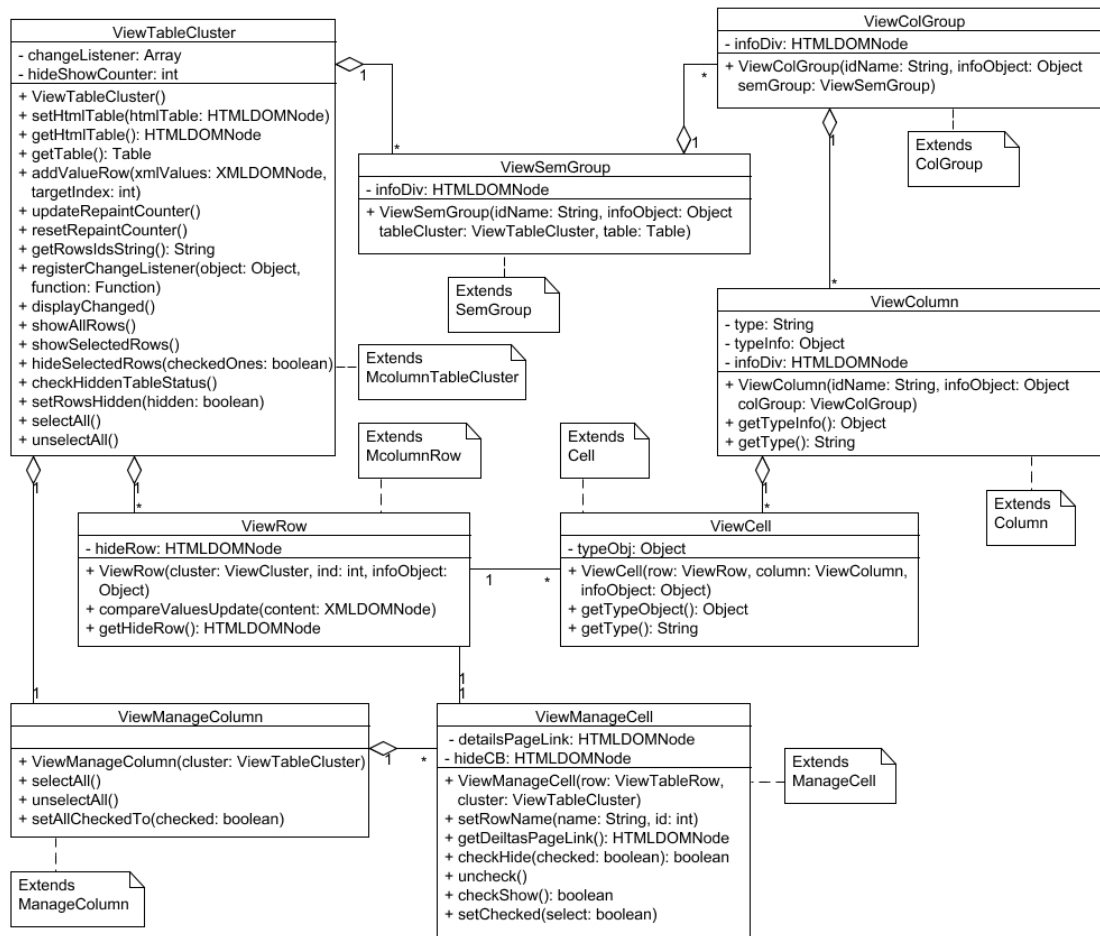


Figure 81: The content of the viewTable package

## tableDisplayController.js

The `tableDisplayController` package contains the implementation for the elements that are used to control the visibility of columns in the view table. The implementation of these objects is split into some base classes and classes that subtype these objects. The base classes are also subtyped by the objects that are used to create the search tables (in the `searchHandlers` package).

For more information see the javadoc like documentation of the classes.

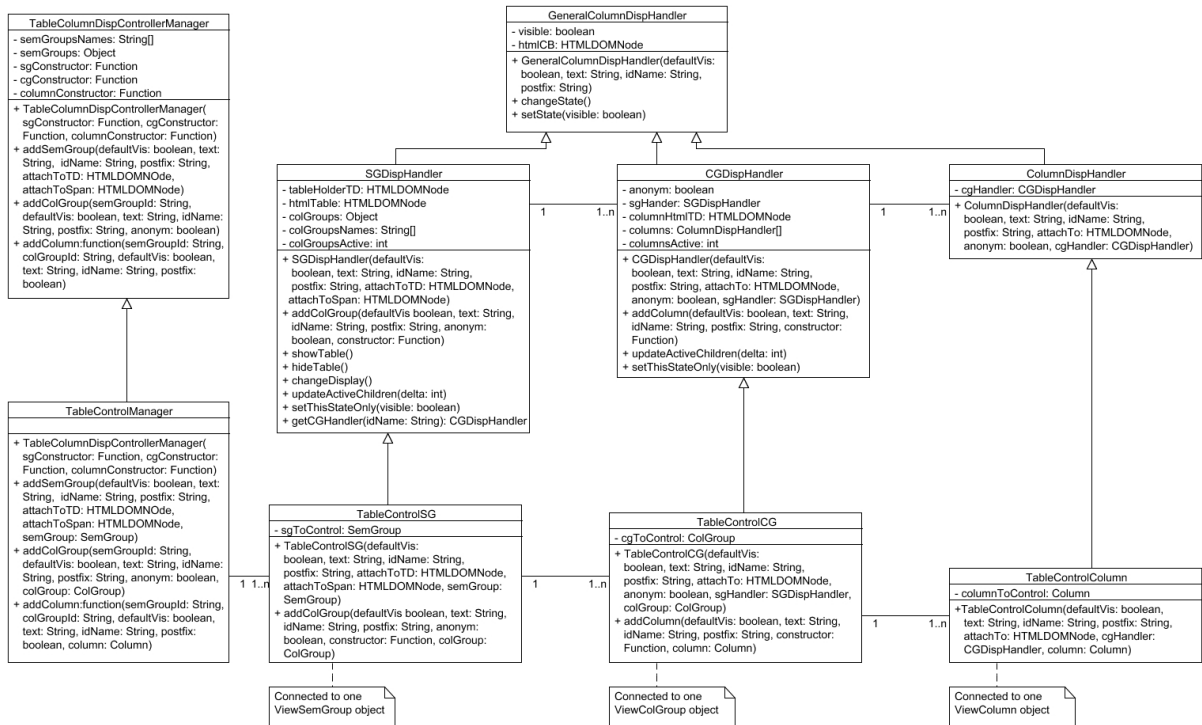


Figure 82: The content of the `tableDisplayController` package

## searchHandlers.js

The `searchHandlers` package contains all classes necessary for the search functionality. This includes the objects for creating the search tables (subtypes of the base classes in the `tableDisplayController` package), an object that manages the existing search tables and the sending of the search requests, and the search tables. The search table is a subtype of the table defined in the `editableTable` package.

For more information see the javadoc like documentation of the classes.







Figure 84: The content of the `exportHandlers` package

## Edit Page

Figure 85 shows all classes available on the “edit page” and their relations. Figure 86 shows for an example table with two rows the used instances of these classes and the relations between these instances. (The instances of the type-objects are not shown.)

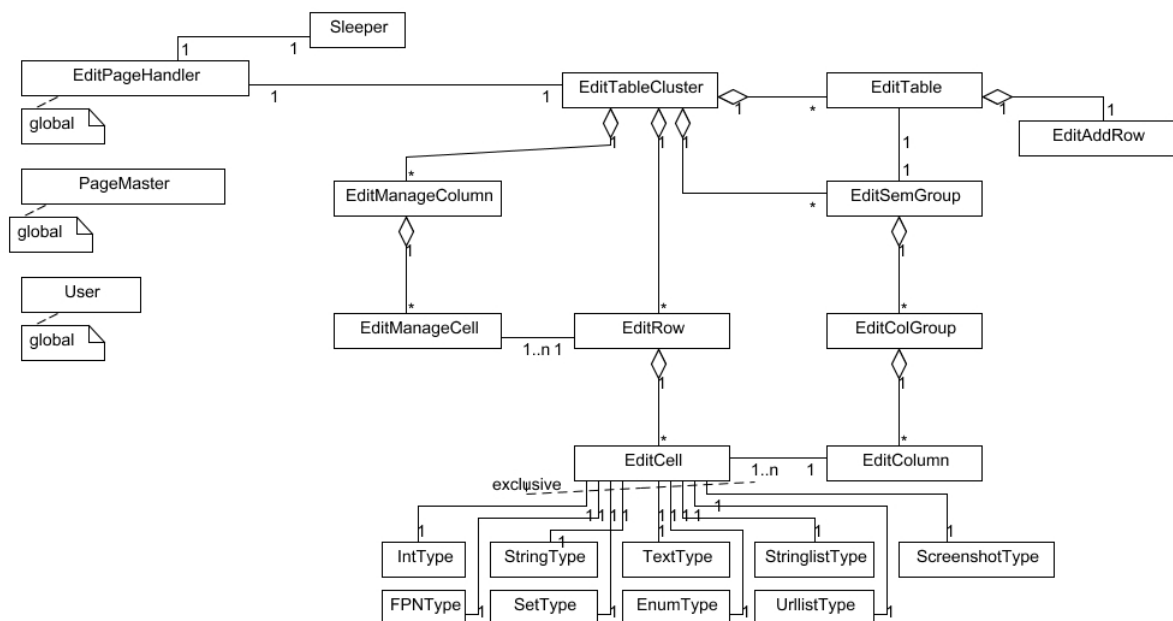


Figure 85: The used classes and their relations on the edit page.

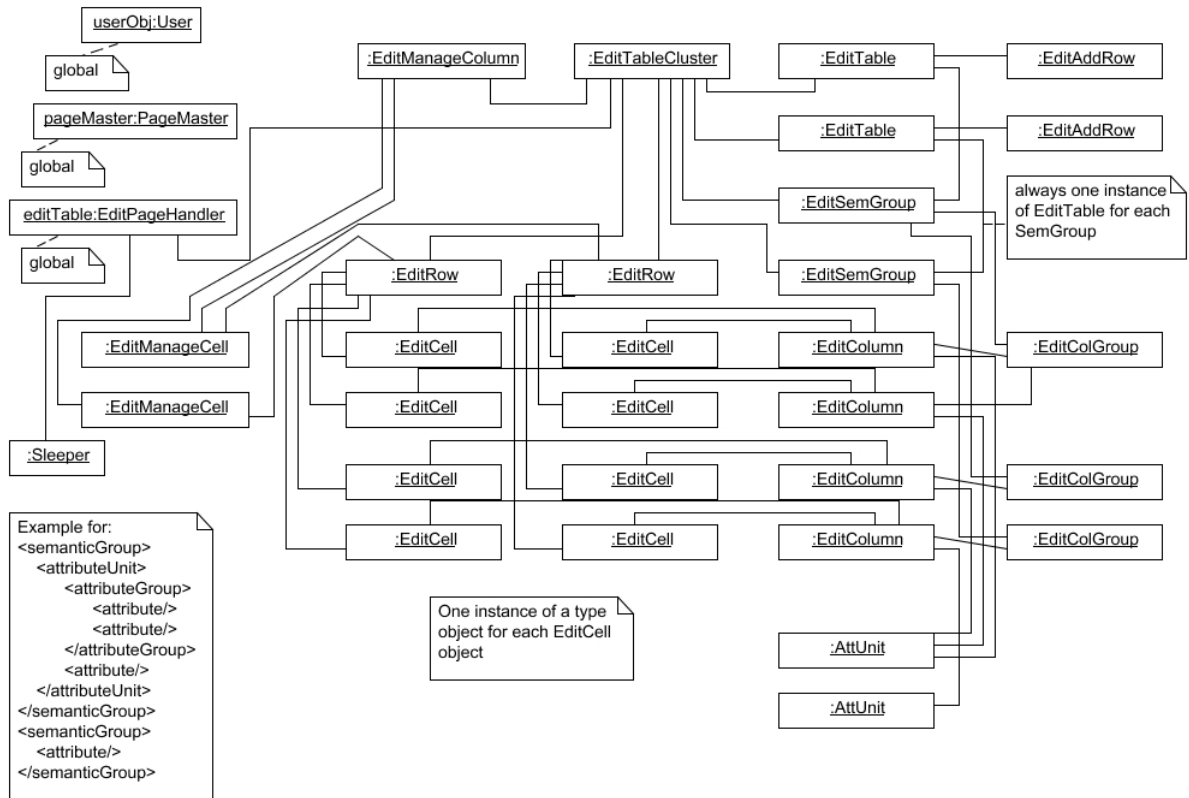


Figure 86: The instances (objects) present on the edit page for managing the shown example table structure

## View Page

Figure 87 shows all classes available on the “view page” and their relations. The classes for the “type-objects” are not shown. To each **ViewCell** object exactly one type-object is attached. Figure 88 shows for an example table with two rows the used instances of these classes and the relations between these instances. These diagrams show the state of the view page if the user has full privileges.



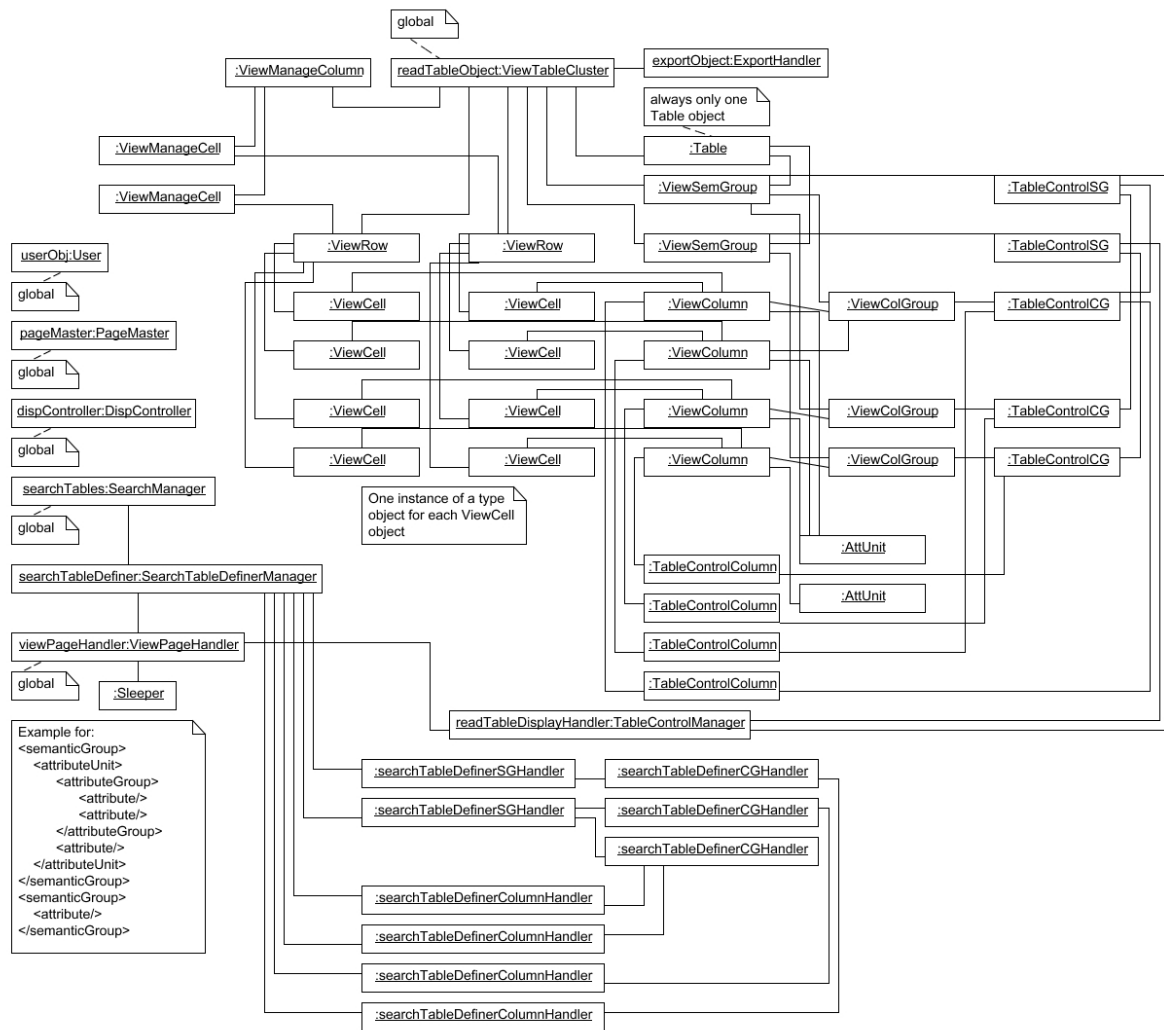


Figure 88: The instances (objects) used on the view page to display/manage the given example table with two rows.

## Client

Figure 89 shows all classes available for the client and their relations. Unlike in the previous figures, also the type hierarchies are shown.

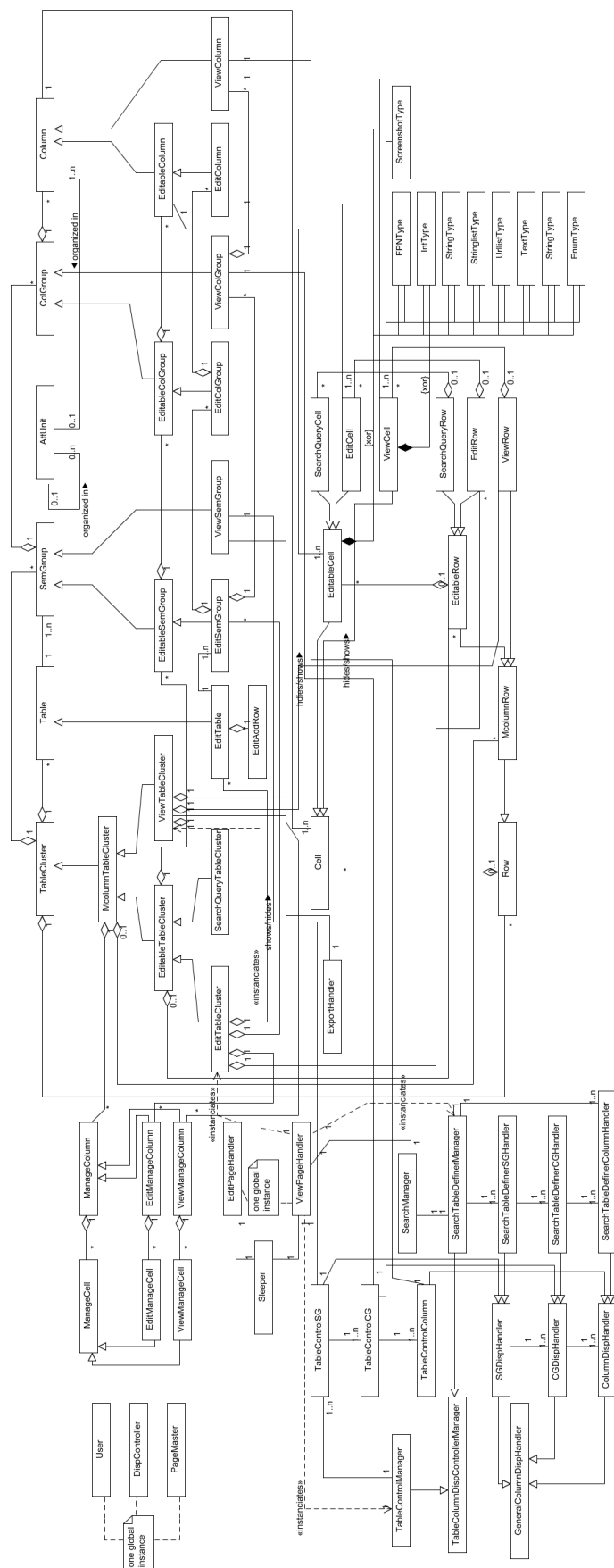


Figure 89: An UML diagram showing all classes available on the client and their relations.

## 11.2 Sequence diagrams and used data formats

This section shows sequence diagrams for the most important and complex sequences. Most of the actions not visualized in this section only consist of a simple method- or function call. All shown sequences include a call to a server script. All parameters of these scripts and all possible return values are given in this section too. All existing server script files are covered by this section.

### 11.2.1 Load index page

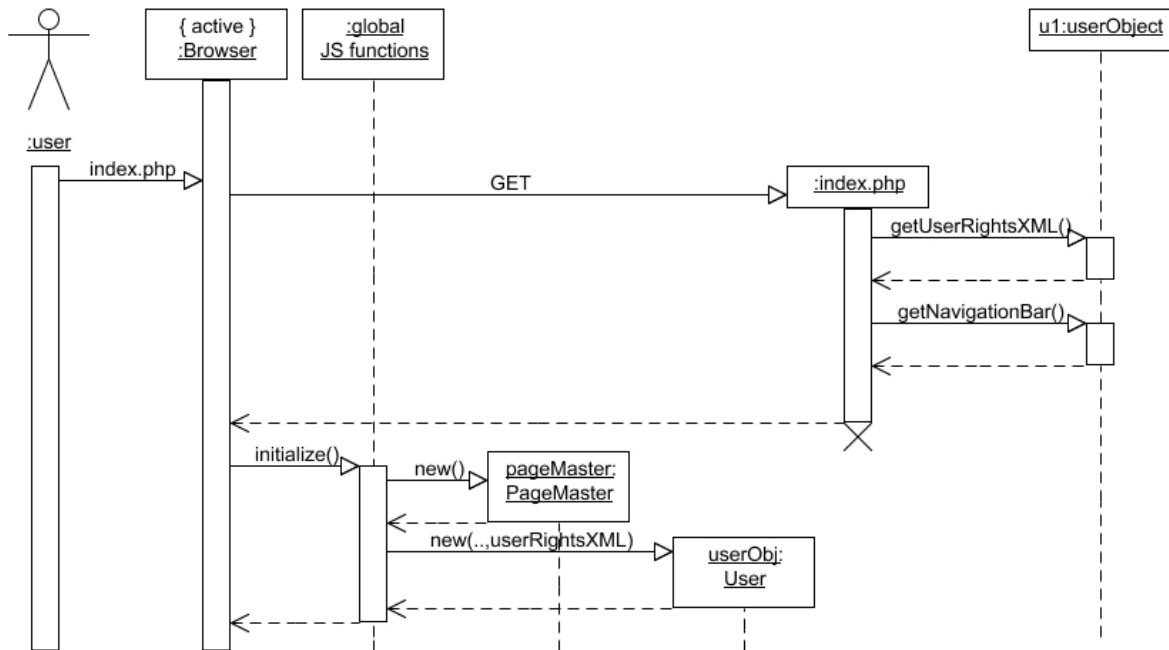


Figure 90: UML diagram showing the loading process of the index page and the initial JavaScript function calls

## 11.2.2 Login

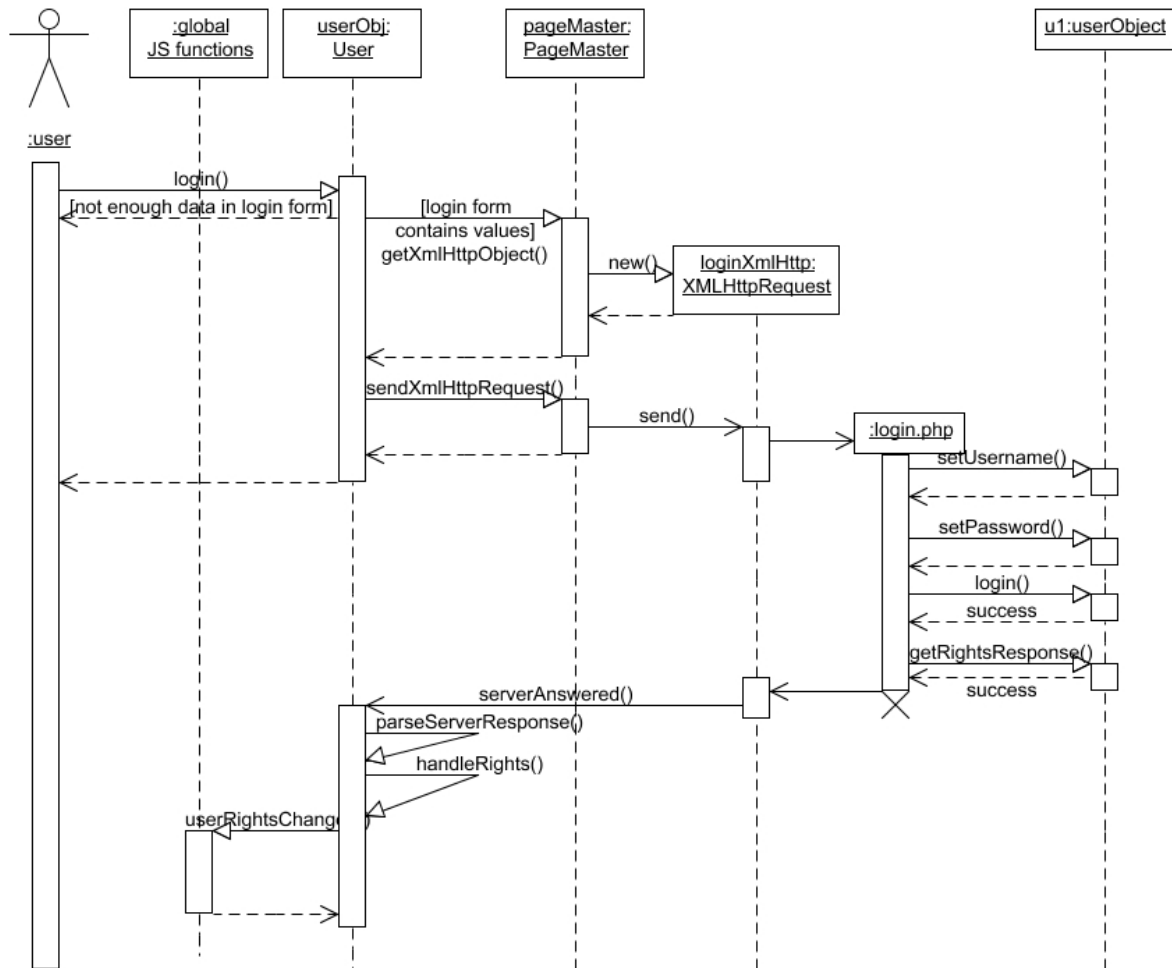


Figure 91: This UML diagram shows the steps of the login procedure after the user clicked onto the “login” link

### login.php: argument/return values:

**request:** POST - data (url encoded):

- sys\_username: containing the username
- sys\_pw: containing the password

Listing 5: XML format of the server response for a successful login

```

<?xml version="1.0" ?>
<loginAnsw status="passed">
  <rights>
    <right>nameOfRight(e.g. "edit")</right>

```



```

    ...
    ...
    </rights>
    <navigation xmlns="http://www.w3.org/1999/xhtml">
      <div id="idOfNavigationElement">content of navigation
        elements</div>
      ...
      ...
    </navigation>
  </loginAnsw>

```

In case of an successful login, the response contains a list with the rights the user has (the `<rights>` element), and a list with the navigation elements (links, the logout form and the name of the currently logged in user). Each navigation element (each `<div>` element being child of the `<navigation>` element) is already in the XHTML namespace and only needs to be inserted as child elements to the navigation bar on the page. The elements are already in the correct order as they shall appear in the navigation bar. All elements already present shall be removed before.

Listing 6: Server response when an error occurred during the login

```

<?xml version="1.0" ?>
<loginAnsw status="failed"/>

```

In case of any failure, for security reasons the response does not contain any information about the case for the error. This includes a not existing username or wrong password.

### 11.2.3 Logout

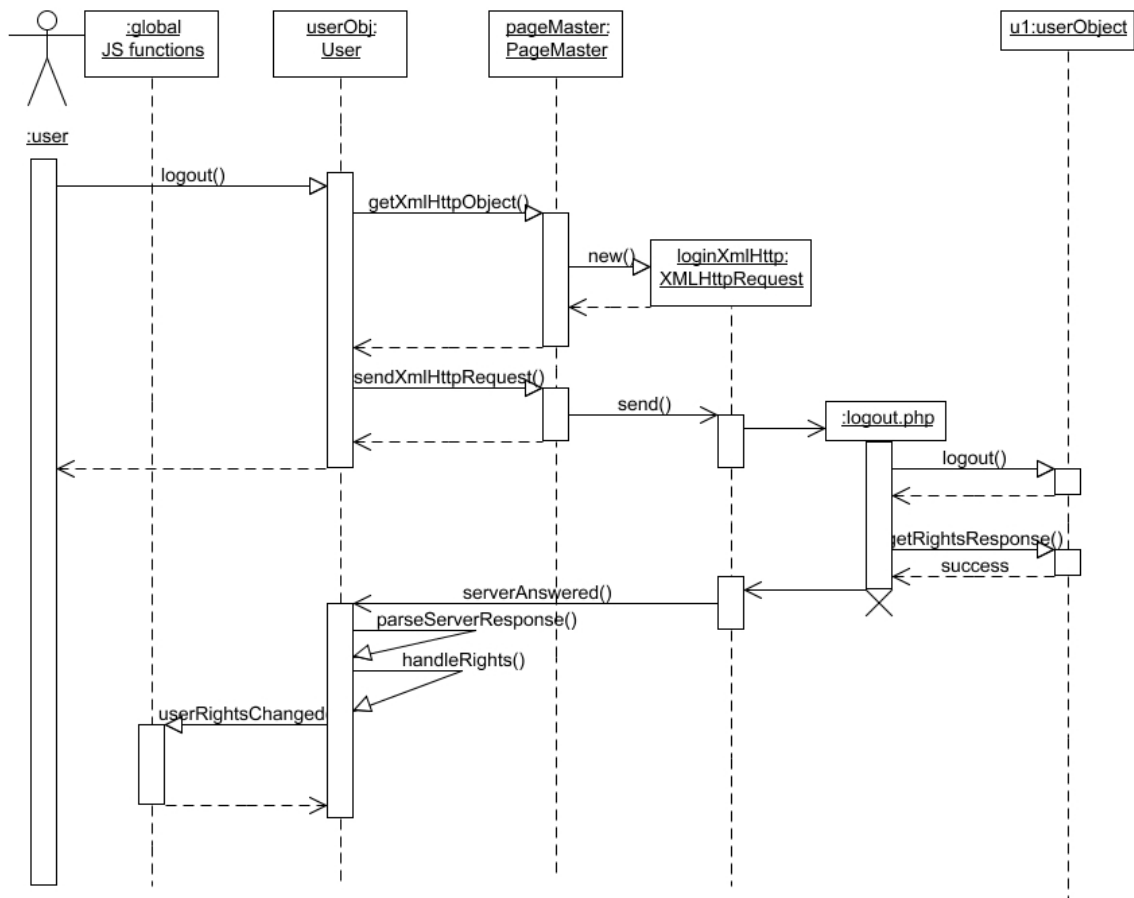


Figure 92: This UML diagram shows the steps in the logout process.

**logout.php: argument/return values:**

**request:** - no arguments needed -

Listing 7: XML format of the server response for a logout-request.

```

<?xml version="1.0" ?>
<loginAnsw status="passed">
  <rights>
    <right>nameOfRight(e.g. "edit")</right>
    ...
    ...
  
```

```

    </rights>
<navigation xmlns="http://www.w3.org/1999/xhtml">
    <div id="idOfNavigationElement">content of navigation
        elements</div>
    ...
    ...
</navigation>
</loginAnsw>

```

Whenever `logout.php` is called, the user is “logged out” afterwards, so no error can appear. This is why the response always has the same content:

A list of the rights a not logged in user has (the `<rights>` element), and a list with the elements that shall appear in the navigation bar (links to the application parts the user may go, or the login form). These elements (the `<div>` children of the `<navigation>` element) are already in the XHTML namespace and the correct order as they shall be inserted.

### 11.2.4 Load the Edit Page

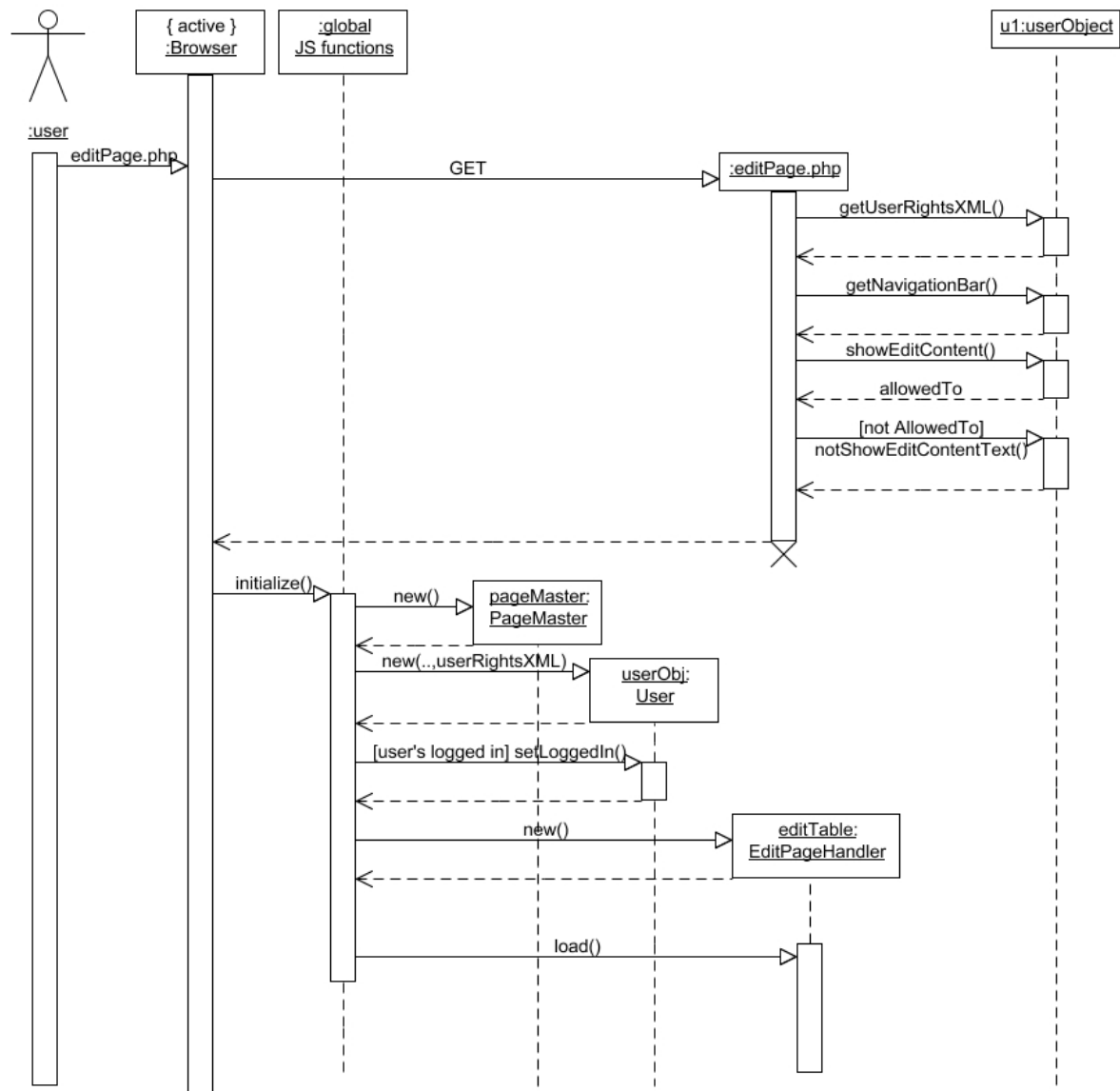


Figure 93: UML diagram showing the loading of the edit page and the function calls to initialize the client.

### 11.2.5 Load Edit Table

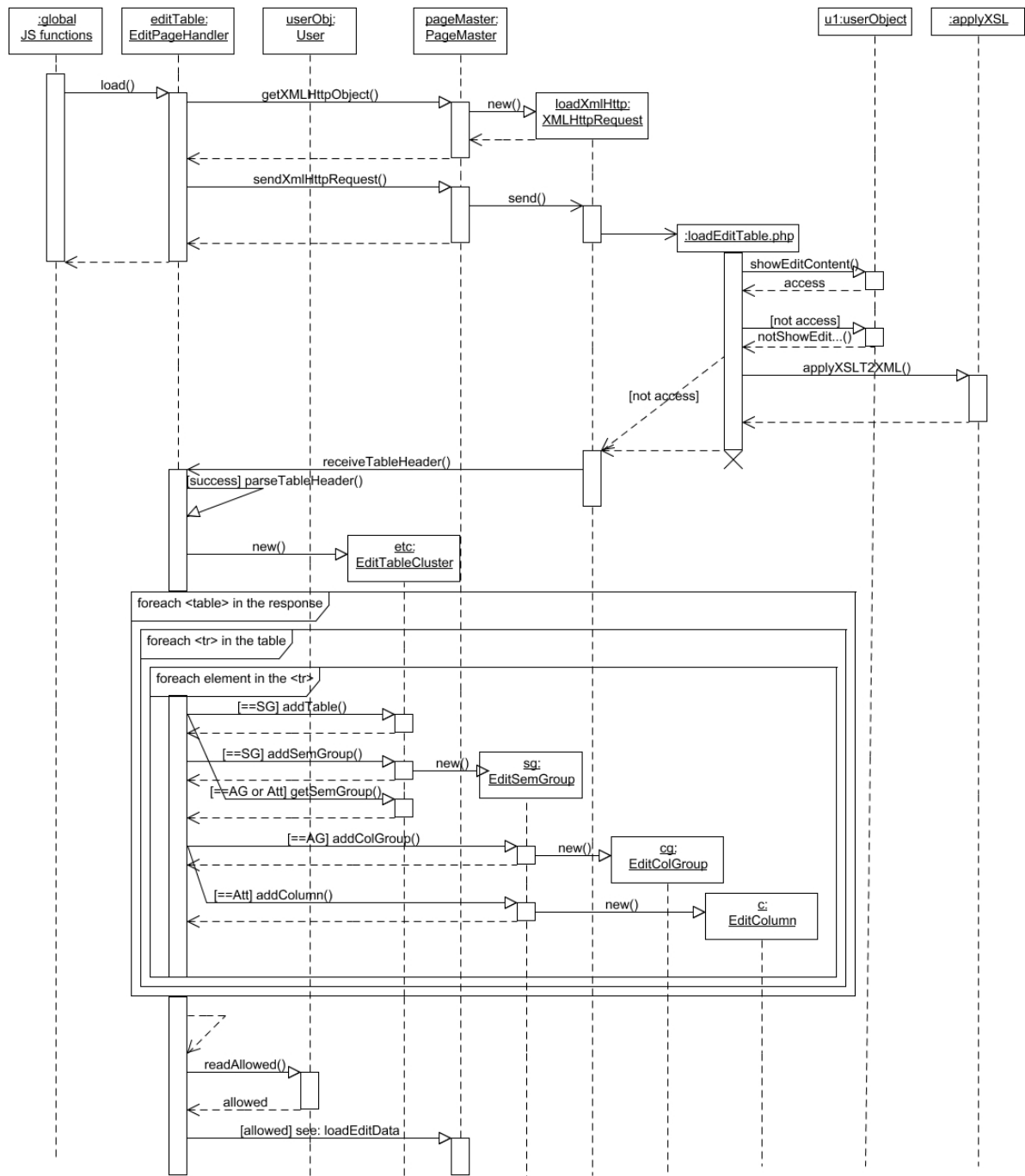


Figure 94: UML diagram showing the loading of the structure of the edit tables and the steps for creating these tables. The loading of the data rows is shown in the next figure.

## loadEditTable.php: argument/return values:

**request:** - no arguments needed -

Listing 8: Server response containing the table structure of the edit tables.

```
<success>
  <div id="tableDiv" xmlns="http://www.w3.org/1999/xhtml">
    <tables id="viewtable">
      <table id="table_"[semanticGroupIdName]>
        <thead id="thead_"[semanticGroupIdName]>
          <col id="col_"[semanticGroupIdName] - [attributeIdName] />
          <tr>
            <th colspan=[x]>
              <info type="semGroup">
                <name>name</name>
                <dbName>id</dbName>
                <description>text</description>
              </info>
              name
            </th>
          </tr>

          <tr>
            <th colspan=[x]>
              <info type="attGroup">
                <semGroup>id of sem group</semGroup>
                <name>name</name>
                <shortName>id</name>
                <description>text</description>
              </info>
              name
            </th>
            ...
          </tr>
        </thead>
        <tbody>
          <tr>
            <th>
              <info type="attribute">
                <semGroup>id of semantic group</semGroup>
                <attGroup>id of attribute group</attGroup>
              </info>
              <name>name</name>
              <shortName>id</shortName>
              <type>type</type>
              <value>
```

```

        <name>value name</name>
        <shortName>value id</shortName>
        <order>integer</order>
    </value>
    ...
</info>
    name
</th>
    ...
</tr>
</thead>
    <tbody/>
</table>
    ....
</tables>
</div>
</success>

```

After the `<info>` elements have been removed, the `<table>` elements can be directly inserted to the page. For each semantic group, one `<table>` exists. The `<info>` elements contains further information about the semantic group/attribute group or attribute: the full name, the id and a description text. The `<info>` elements of attribute groups also contain the id of the semantic group they belong to. For anonymous attribute groups, the `<shortName>` (!) element contains “sys\_anonym”, and the `<name>` element (!) contains the id of the attribute group (consisting of {name of the attribute}\_aag”). The `<info>` elements of the attributes further contain the id of the attribute group they belong to, and about their data type (copied from the table structure XML file) The `<value>` elements are only present if the type is “set” or “enum”.

**response(failure):** *If the user has no rights to access the edit page, the following response is sent:*

Listing 9: Server response of `loadEditTable.php` if the user is not allow to enter this page.

```

<?xml version="1.0" ?>
<error reason="access denied">
    <span xmlns="http://www.w3.org/1999/xhtml">
        <h1>Access denied</h1>
        <h2>You don't have the rights to enter the user management
            center.</h2>
        <h2>You might have to log in first?</h2>
    </span>
</error>

```

`<h2>You might have to log in first?</h2>` is only sent if there's no user logged in.

### 11.2.6 Load Edit Data

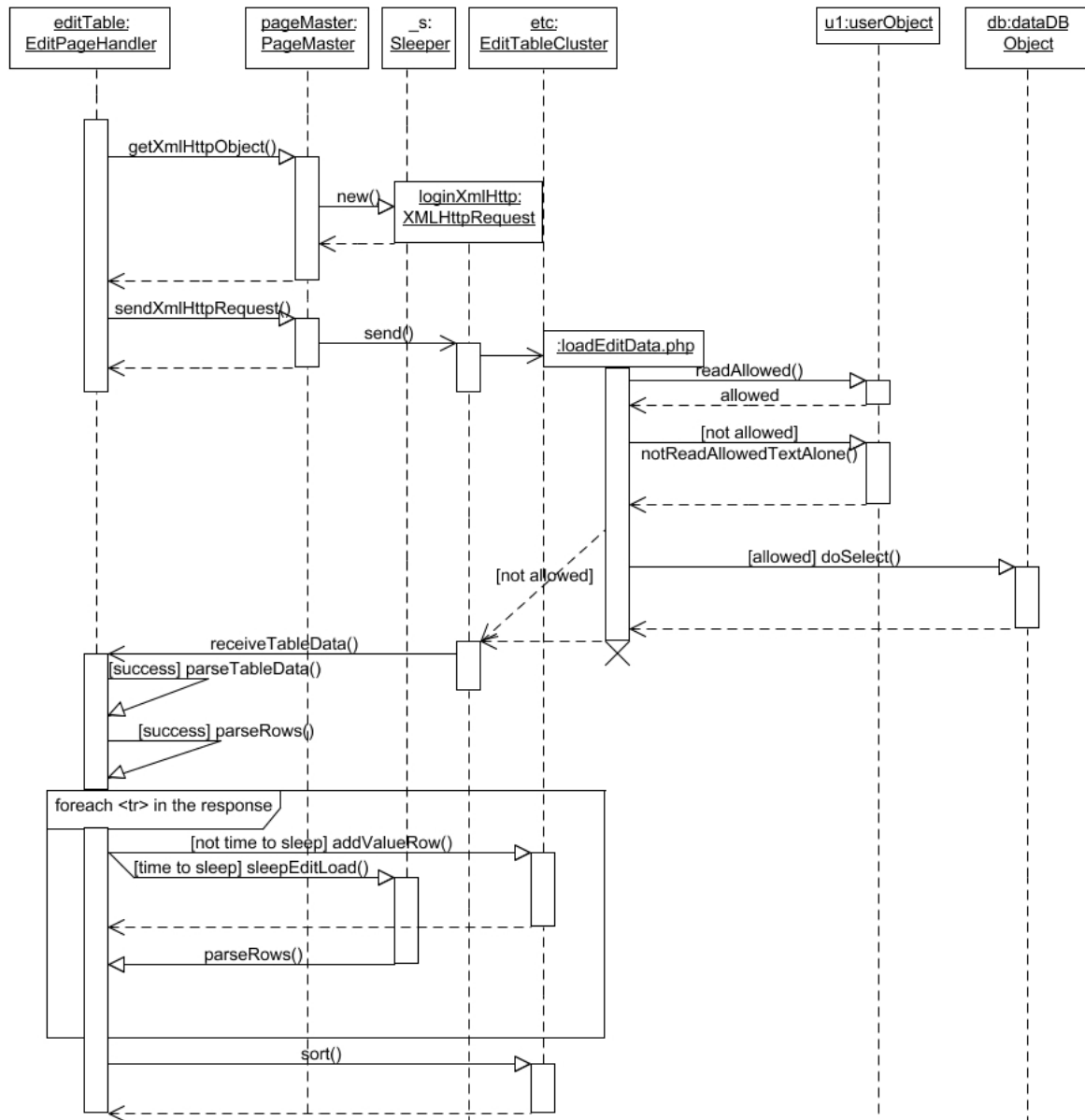


Figure 95: UML sequence diagram showing the loading of rows for the edit tables and how they are inserted into these tables.

#### loadEditData.php: argument/return values:

**request:** POST - data (url encoded):

- *sys\_semGroups*: a semicolon separated list of semantic group ids (= database table names), that defines from which tables data shall be loaded
- *[semGroup]*: for each entry in the list in *sys\_semGroups*, there must be a POST argument with the id of the semantic group as name. Each of these entries contains



a semicolon separated list of fields that shall be selected from the corresponding database table. “\*” as wildcard for all fields is allowed.

**response(success):** The response consists of an XML document of the form shown below. For each row, each attribute can be uniquely identified by the combination of semantic group and attribute id. This is the reason for the attributes being grouped by the <semanticGroup> elements. So they can be easily accessed by the client.

Listing 10: Response of the loadEditData.php script containing the entries stored on the server.

```
<?xml version="1.0" ?>
<data numb=[numbResults]>
  <tr techId=[idOfTechnique1] techName=[nameOfTechnique1]>
    <semanticGroup tName=[idOfSemanticGroup1]>
      <attribute>
        <shortName>uniqueNameOfAttribute1</shortName>
        <value>valueofAttribute1InRow1</value>
      </attribute>
      <attribute>
        <shortName>uniqueNameOfAttribute2</shortName>
        <value>valueofAttribute2InRow1</value>
      </attribute>
      ...
    </semanticGroup>
    <semanticGroup tName="idOfSemanticGroup2">
      ...
    </semanticGroup>
  </tr>
  <tr techId="idOfTechnique2" techName="nameOfTechnique2">
    ...
  </tr>
  ...
</data>
```

**response(failure):** *No POST-arguments founds:*

Listing 11: Error-response of the loadEditData.php script if no or not enough POST-data were submitted.

```
<?xml version="1.0" ?>
  <error reason="no data">
    <span xmlns="http://www.w3.org/1999/xhtml"><h1>ERROR</h1><h2>
      No data specifying the request found</h2></span>
    </error>
```

*The user does not have the read right:*

Listing 12: Error-response of the loadEditData.php script if the user is not allowed to read the stored data.

```
<?xml version="1.0" ?>
<error reason="access denied">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>Access denied</h1>
    <h2>You don't have the rights to perform any operations on
      the data</h2>
    <h2>You might have to log in first?</h2>
  </span>
</error>
```

<h2>You might have to log in first?</h2> is only shown if no user is logged in.

### 11.2.7 Load Screenshot

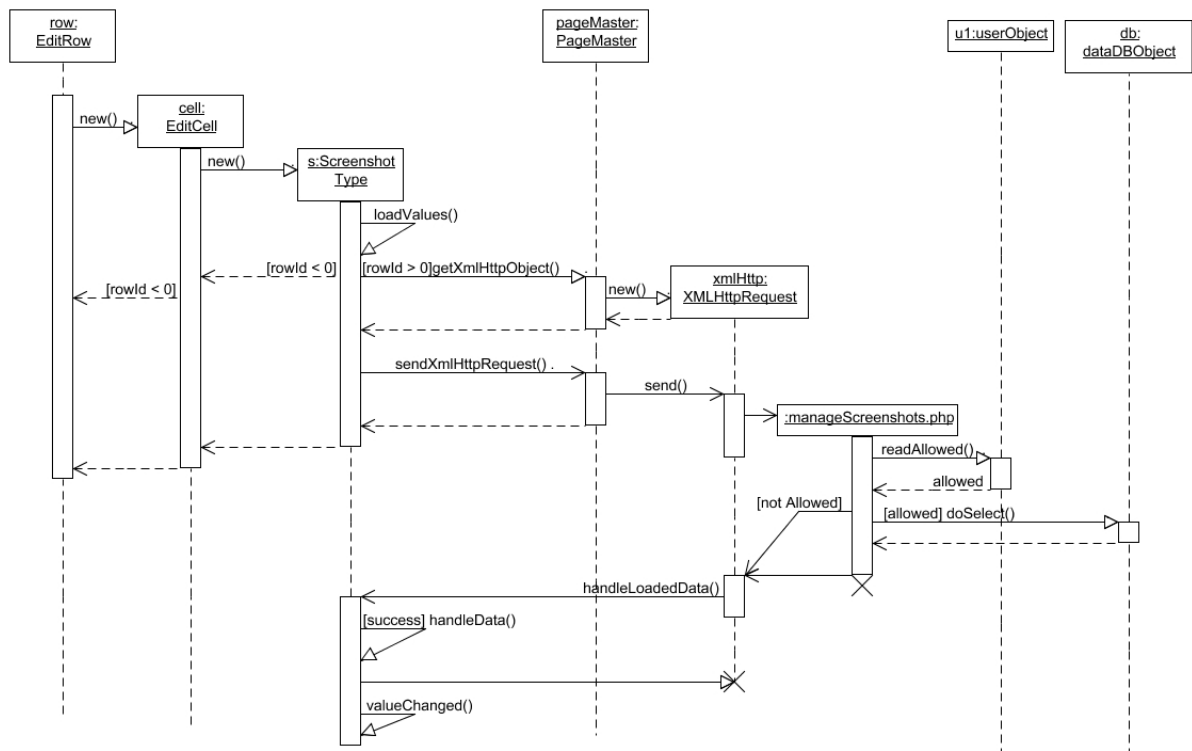


Figure 96: UML sequence diagram showing the loading process for all screenshots belonging to a row.

For a description of the argument and return values of the manageScreenshots.php file, please see section “Screenshots”.

## 11.2.8 Save

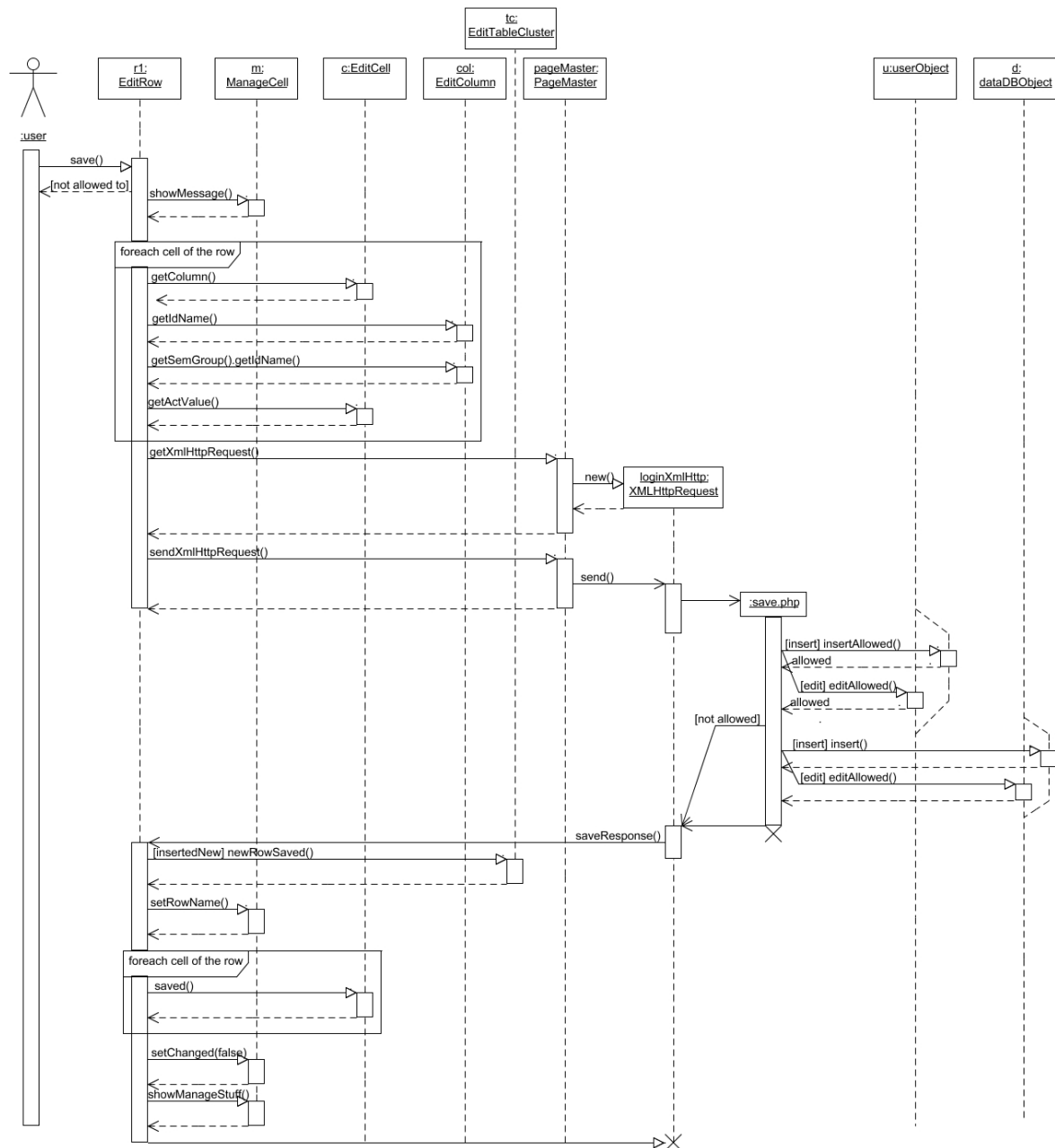


Figure 97: UML sequence diagram showing the necessary steps for saving an entry.

**save.php: argument/return values:**

**request:** POST - data (url encoded):

- *sys\_id*: the id of the entry to update, < 0 if a new entry shall be saved
- *sys\_semGroups*: a semicolon separated list of the ids of all semantic groups (= the names of all database tables) affected by the operation. If a new entry shall be inserted, the first entry in the list must be the database table whose id field has the

“auto\_increment” property. For updating existing rows, the order of the names is irrelevant.

- *[semGroup]*: For each entry in the list defined in sys\_semGroups, there must be one POST argument having the name of the semantic group that contains a semicolon separated list of the ids of the attributes within this semantic group (= the fields of the according database table) that are affected by the save operation.
- *[semGroup]\_[attribute]*: For each attribute in any of the lists defined in the [sem-Group] fields, there must be a POST argument whose name is build using the id of the semantic group followed by “\_” and the id of the attribute. This field must contain the value that shall be written to the database for this field.

EXAMPLE (as members of the \$\_POST array that is provided by php, all urlencoded)

```
$_POST["sys.id"] = -1
$_POST["sys.semGroups"] = "table1;table2;table3;"
$_POST["table1"] = "att1;att2;"
$_POST["table2"] = "att1;"
$_POST["table3"] = "att1;"
$_POST["table1_att1"] = "value1"
$_POST["table1_att2"] = "value2"
$_POST["table2_att1"] = "value3"
$_POST["table3_att1"] = "value4"
(corresponds to:
sys_id=-1&sys_semGroups=table1;table2;table3;&
table1=att1;att2;&...&table1_att1=value1&....)
```

Listing 13: Response of the save.php script when saving the entry was successful.

```
<?xml version="1.0" ?>
  <success>
    <id>[id of saved technique]</id>
    <name>[name of saved technique]</name>
  </success>
```

**response(failure):** *Not enough rights to save the data:*

Listing 14: Response of the save.php script if the user is not allowed to save the data.

```
<?xml version="1.0" ?>
<error reason="access denied">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ACCESS DENIED</h1><h2>You have no rights to access this
      functionality </h2>
    <h2>You may log in first... </h2>
  </span>
</error>
```

<h2>You may log in first...</h2> is only shown if no user is logged in. *Problems saving data:*

Listing 15: Response of the save.php script if an error occurred while saving the data.

```
<?xml version="1.0" ?>
<error reason="[int code of error defined by mysql database]">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>[int code of error]</h1><h2>textual description of
      error</h2>
  </span>
</error>
```

### 11.2.9 Add Screenshot

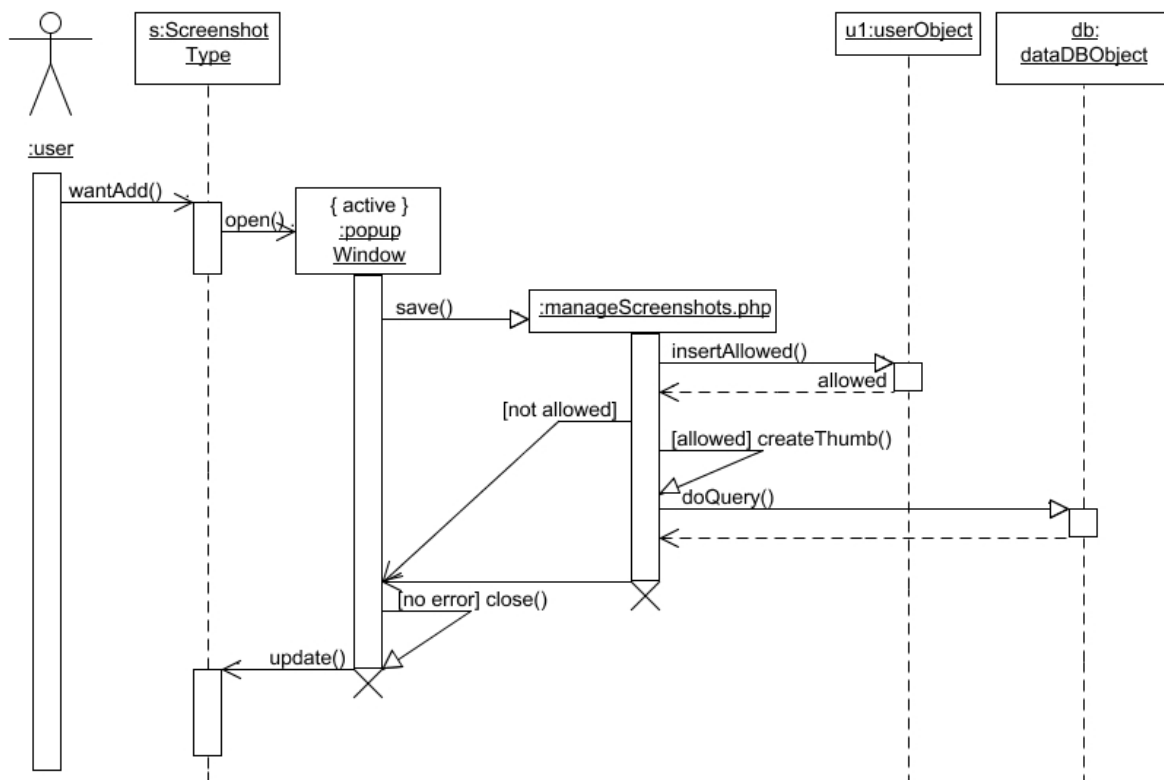


Figure 98: UML sequence diagram showing the steps necessary to save a screenshot.

For a description of the argument and return values of the *manageScreenshots.php* file, please see “Screenshots”.

## 11.2.10 Delete

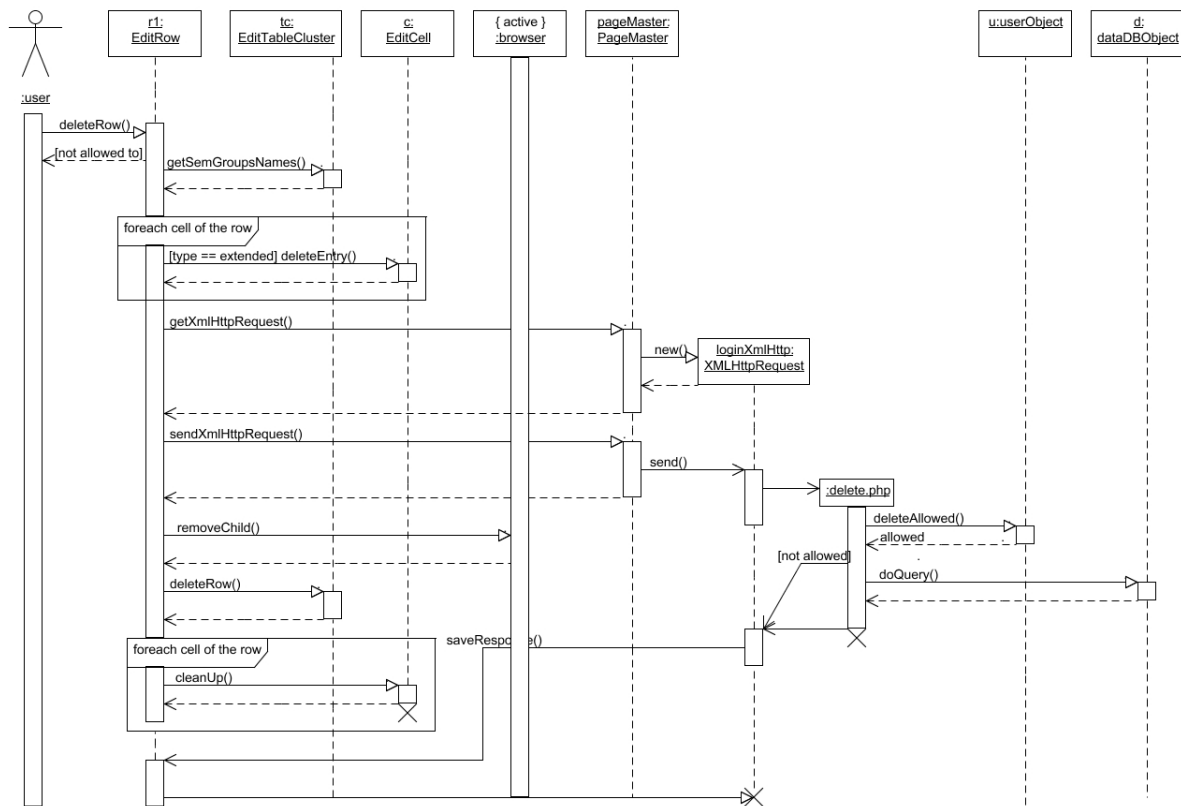


Figure 99: UML sequence diagram showing the deletion process of an entry.

**delete.php: argument/return values:**

**request:** POST - data (url encoded):

- *sys\_action*: must contain the string "delete"
- *sys\_id*: the id of the entry that shall be deleted
- *sys\_semGroups*: a semicolon separated list containing the names of the database tables (= ids of the according semantic groups) the entry with the specified id shall be deleted from.

Listing 16: Response of the delete.php script if deleting the entry was successful.

```

<?xml version="1.0" ?>
<success></success>
  
```

**response(failure):** *The user does not have the right to delete the entry:*

Listing 17: Response of the delete.php script if the user is not allowed to delete the entry.

```
<?xml version="1.0" ?>
<error reason="access denied">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ACCESS DENIED</h1>
    <h2>You have no rights to access this functionality</h2>
    <h2>You may log in first...</h2>
  </span>
</error>
<h2>You may log in first...</h2> is only shown if no user is logged in.
```

*The sys\_semGroups argument contains a name that is not a name of a semantic group defined in the current table structure XML file (specified by the URL parameter cfg\_pref):*

Listing 18: Response of the delete.php script if the request contains a table it is not allowed to change.

```
<?xml version="1.0" ?>
<error reason="critical error">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>CRITICAL ERROR</h1>
    <h2>Your client software sent an request that lead to an
      error. Please contact us!</h2>
  </span>
</error>
```

This error cannot appear if the unchanged JavaScript client is used, but only if someone tries to access database tables s/he is not allowed to.

*problems while saving the data:*

Listing 19: Response of the delete.php script if an error occurred.

```
<?xml version="1.0" ?>
<error reason="database error">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>integer error code as provided by mysql</h1>
    <h2>textual description of the error</h2>
  </span>
</error>
```

*no POST data sent:*

Listing 20: Response of the delete.php script if no or not enough post data were sent.

```
<?xml version="1.0" ?>
<error reason="missingData">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ERROR</h1>
    <h2>no data sent</h2>
  </span>
</error>
```

### 11.2.11 Delete Screenshot

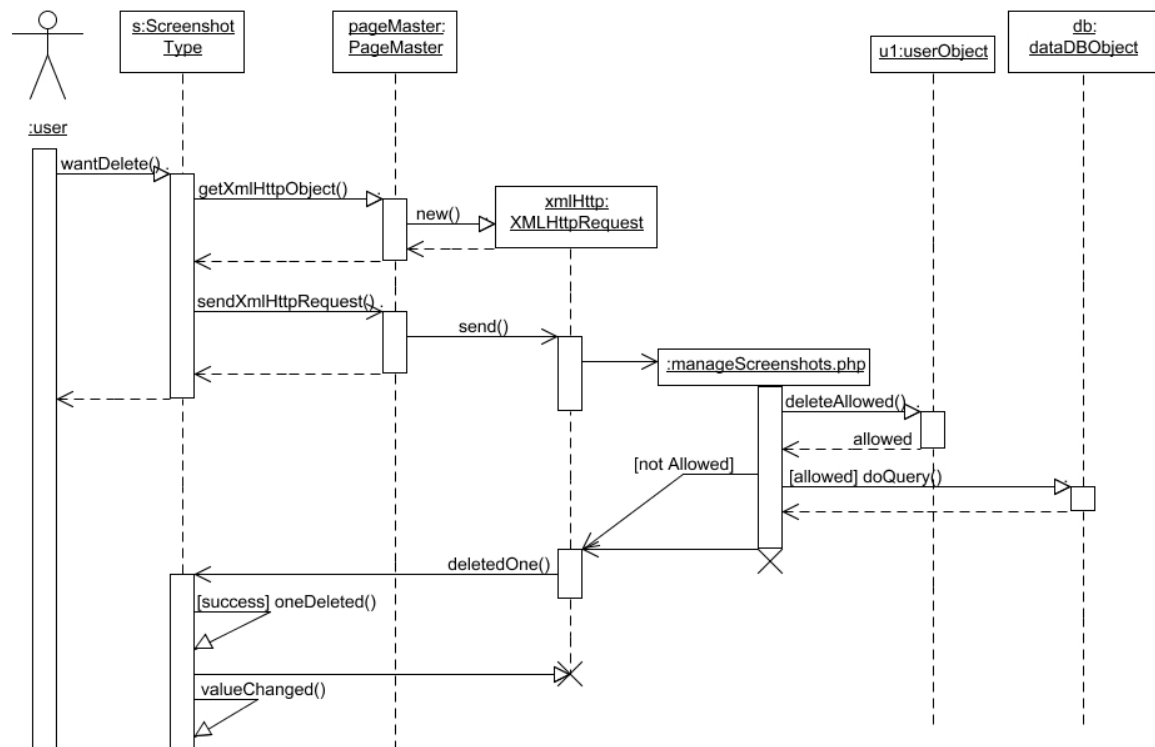


Figure 100: UML sequence diagram for deleting a screenshot.

For a description of the argument and return values of the manageScreenshots.php file, please see “Screenshots”.



### 11.2.12 Load the View Page

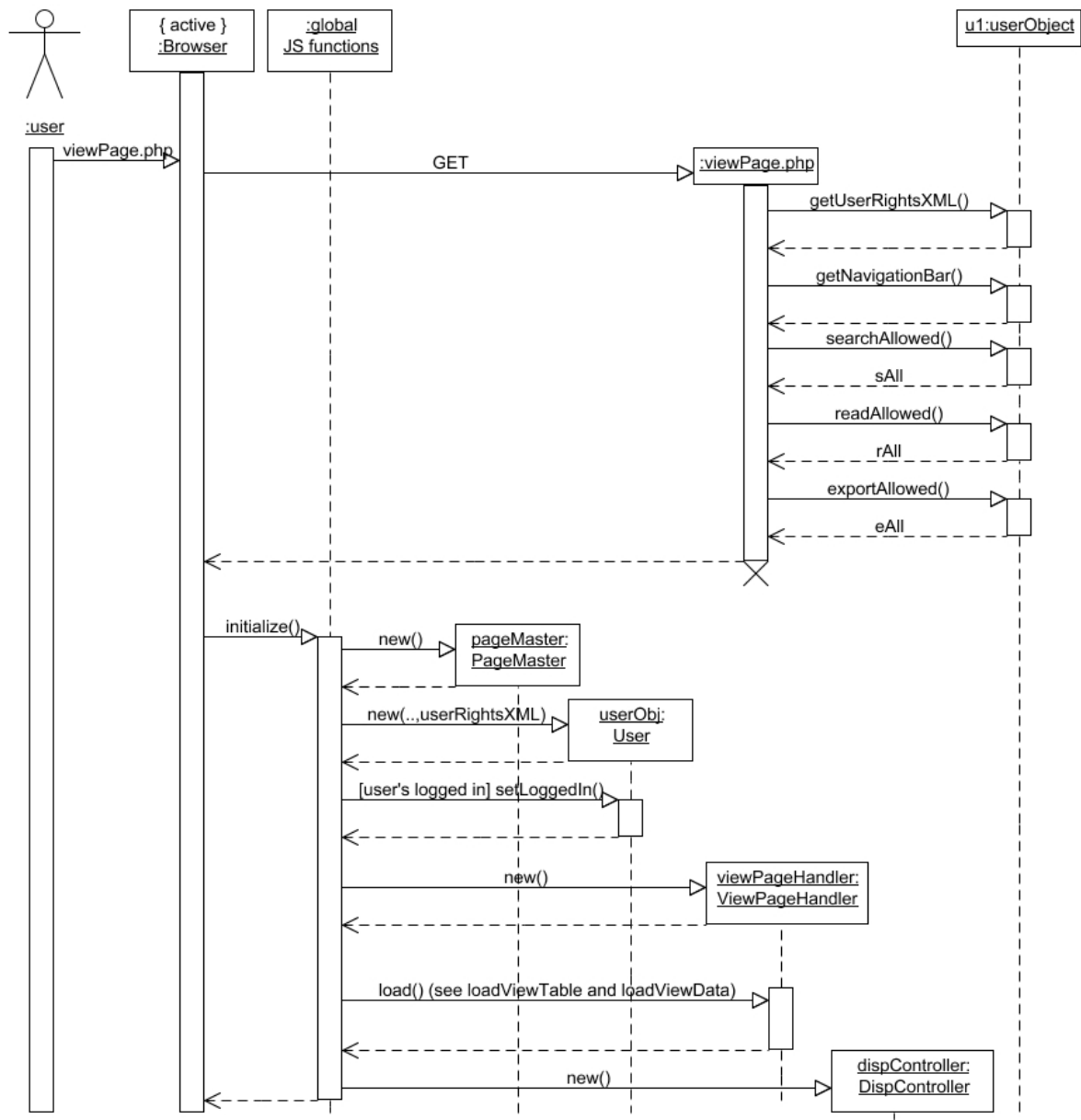


Figure 101: UML sequence diagram of the loading of the view page and the first JavaScript calls to initialize the client.

Figure 102: UML sequence diagram of the loading of the table structure of the view table. The loading of the entries is shown in the next diagram.



**loadViewElements.php: argument/return values:**

**request:** POST - data (url encoded):

- *sys\_part*: must contain the string “header”

Listing 21: Response of the loadViewTable.php script containing information about the table structure of the view table.

```
<success xmlns="http://www.w3.org/1999/xhtml">
  <tables>
    <table id="viewtable">
      <thead id="theadview">
        <col id="col_"[semanticGroupIdName]_[attributeIdName] />
      <tr>
        <th colspan=[x]>
          <info type="semGroup">
            <name>name</name>
            <dbName>id</dbName>
            <description>text</description>
          </info>
          name
        </th>
        ...
      </tr>
      <tr>
        <th>
          <info type="attUnit">
            <name>name of the att unit</name>
            <shortName>id of the att unit</shortName>
            <description>description text</description>
            <attributes>
              <attribute>
                <shortName>id of attribut</shortName>
                <attGroup>id of attGroup</attGroup>
                <semGroup>id of semGroup</semGroup>
              </attribute>
              ...
            </attributes>
          </info>
          name of the att unit
        </th>
      </tr>
      ...
    <tr>
      <th colspan=[x]>
        <info type="attGroup">
          <semGroup>id of sem group</semGroup>
```

```

        <name>name</name>
        <shortName>id</name>
        <description>text</description>
    </info>
    name
</th>
...
</tr>
<tr>
    <th>
        <info type="attribute">
            <semGroup>id of semantic group</semGroup>
            <attGroup>id of attribute group</attGroup>
            <name>name</name>
            <shortName>id</shortName>
            <type>type</type>
            <value>
                <name>value name</name>
                <shortName>value id</shortName>
                <order>integer</order>
            </value>
            ...
        </info>
        name
    </th>
    ...
</tr>
</thead>
<tbody/>
</table>
.....
</tables>
</success>

```

The table head consists of the following rows:

One row with the cells and names for the semantic groups. Then one row for each depth of attribute units, followed by one row with the cells for the attribute groups and one row with the cells for the attributes.

After the `<info>` elements have been removed, the `<table>` element can be directly inserted to the page. The `<info>` elements contains further information about the semantic-/attribute group, attribute unit or attribute:

the full name, the id and a description text.

The `<info>` elements of attribute units further contain a list of attributes that are grouped by this attribute unit. For each of these attributes, their id as well as the id of the semantic- and attribute group they belong to are given. For anonymous attribute units, the `<name>` element is empty, and the `<shortName>` element contains the string "sys\_anonym".

The `<info>` elements of attribute groups also contain the id of the semantic group they belong to. For anonymous attribute groups, the `<shortName>` (!) element contains

“sys\_anonym”, and the `<name>` element (!) contains the id of the attribute group (consistent of {name of the attribute}”\_aag”).

The `<info>` elements of the attributes further contain the id of the attribute group they belong to, and about their data type (copied from the table structure xml file) The `<value>` elements are only present if the type is “set” or “enum”.

**response(failure):** *no sys\_part argument:*

Listing 22: Response of the loadViewTable.php script if the sys\_part argument is not defined.

```
<?xml version="1.0" ?>
<error reason="not_enough_data_sent(1)">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ERROR</h1>
    <h2>Data Missing</h2>
  </span>
</error>
```

*no rights:*

Listing 23: Response of the loadViewTable.php file if the user is not allowed to read the entries.

```
<?xml version="1.0" ?>
<error reason="access denied">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ACCESS DENIED</h1>
    <h2>You have no rights to access this functionality</h2>
    <h2>You may log in first...</h2>
  </span>
</error>
```

`<h2>You may log in first...</h2>` is only shown if no user is logged in.

### 11.2.14 Load View Data

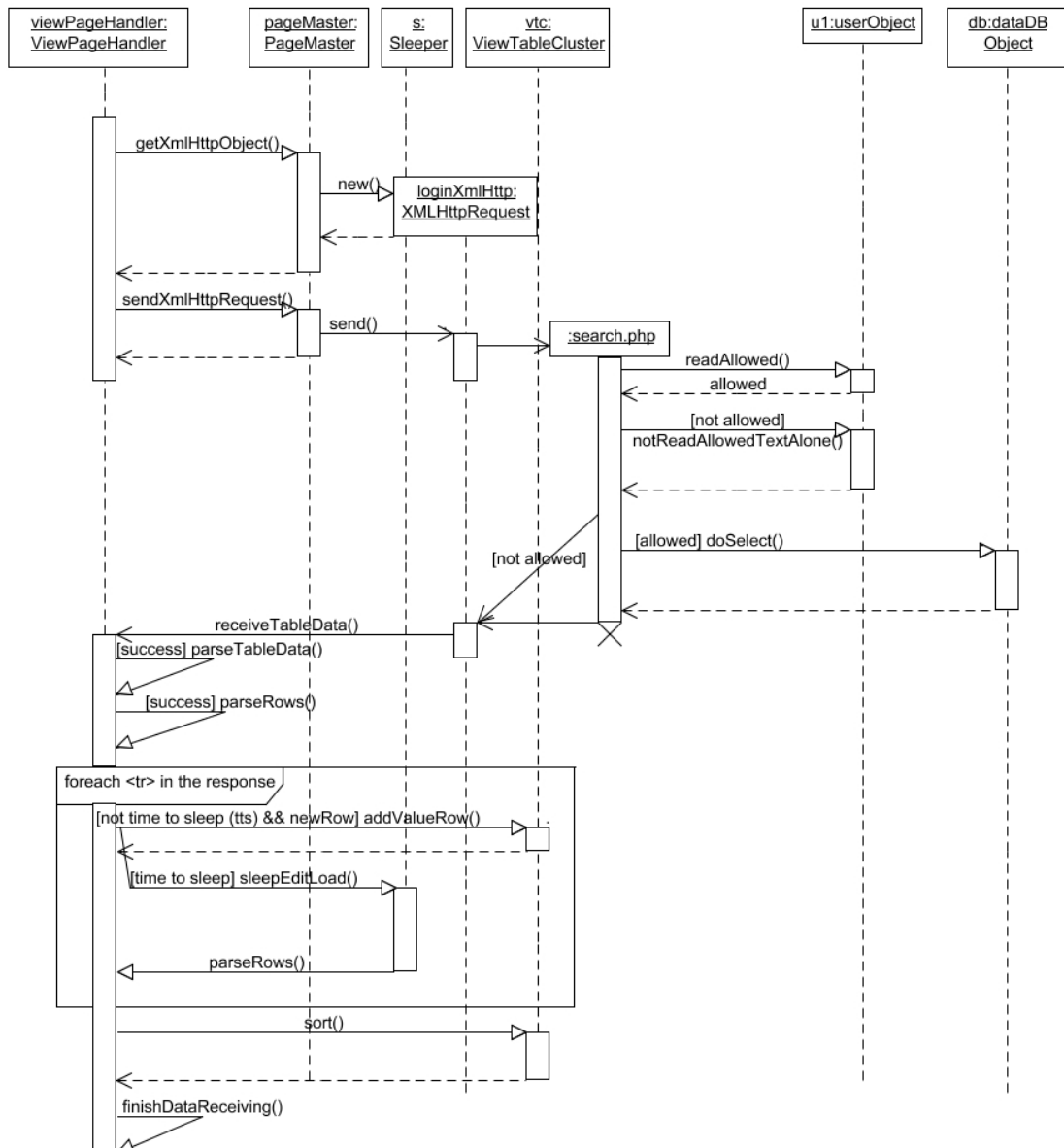


Figure 103: UML sequence diagram showing the loading of rows for the view table.

For a description of the argument/return values of *search.php*, see the next section.

### 11.2.15 Search

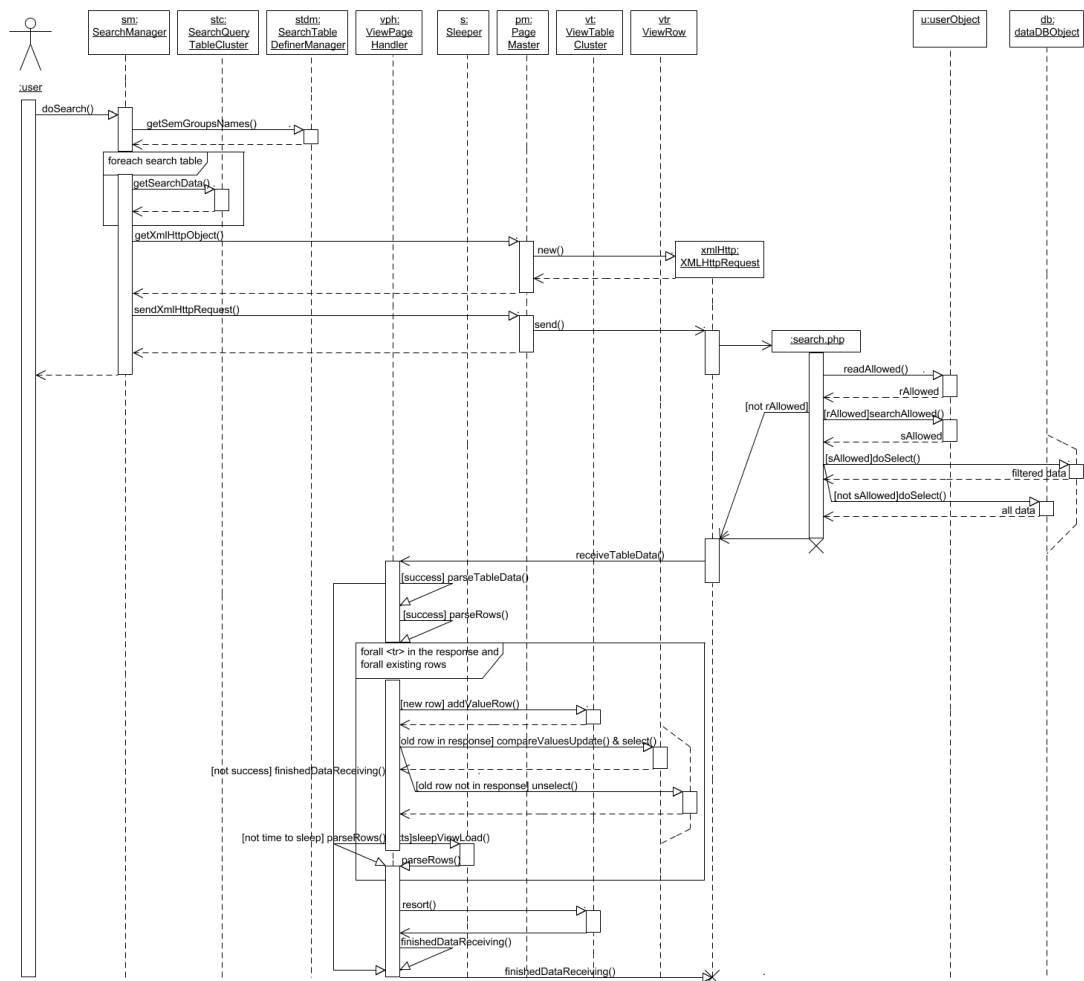


Figure 104: UML sequence diagram showing the steps of the search functionality. (Without the creation of the search tables.)

#### search.php: argument/return values:

**request:** POST - data (url encoded):

- *sys\_semGroups*: a semicolon separated list of the ids of the semantic groups (= names of the database tables) from which attributes (fields) shall be displayed. (not the attributes for which constraints were defined by the user, but the ones that shall be in the output)
- *[semGroups]*: For each entry in sys\_semGroups, an argument with the id of the semantic group as name must exist that contains a semicolon separated list of database table field names (= id of attributes) that shall be selected from the database table.
- *sys\_numbTable*: must contain the number of search tables that were used to define the search constraints

- *sys\_semGroups[tNumb]*: for each search table (tNumb = 1... sys\_numbTables), a semicolon separated list with the names of the database tables that contain fields for which constraints has been defined by this search table must be given.
- *[semGroups]\_[tNumb]*: for each database table defined by an entry in one of the sys\_semGroups[tNumb] arguments (tNumb = 1 ... sys\_numbTables) described above, a semicolon separated list with the names of the fields for which constraints exist must be given.
- *sys\_numbRows[tNumb]*: the number of rows the searchTable “tNumb” has (always one in the actual configuration).
- *[semGroup]\_[field][tNumb]\_[rNumb]*: the value (=constraint) for the field [field] in the database table [semGroup], belonging to the [tNumb] searchTable on row [rNumb] (So each value submitted by the client can be identified by the table, the field, and number of search table and row)

#### EXAMPLE:

assume a database table human(id,name,homeCountry,motherTongue,size), and two search tables st1(name,homeCountry) and st2(name,size), each having one search row.

st1:(‘anonym’,‘Austria’), st2:(‘anonym’,180).

Assuming that all fields of the DB entries that fit the query should be transmitted, the search tables would be encoded like this:

sys\_semGroups = human

human = id;name;homeCountry;motherTongue;size (or human = \* , as these values are only transformed to the <what> part in ”SELECT <what> FROM ...”)

sys\_numbTables = 2

sys\_numRows1 = 1

sys\_numRows2 = 1

sys\_semGroups1 = human

human1 = name;homeCountry

sys\_semGroups2 = human

human2 = name;size

human\_name1.1 = anonym

human\_homeCountry1.1 = Austria

human\_name2.1 = anonym

human\_size2.1 = 180

Listing 24: Response of the search.php file containing the results of the search.

```
<?xml version="1.0" ?>
<data numb="<numbResults>">
  <tr techId="id1" techName="name1">
    <semanticGroup tName="idNameOfTable1">
      <attribute>
        <shortName>idNameOfField1</shortName>
        <value>VALUE</value>
      </attribute>
```



```

        <attribute>
            <shortName>idNameOfField2</shortName>
            <value>VALUE</value>
        </attribute>
        ...
    </semanticGroup>
    <semanticGroup tName="idNameOfTable2">
        <attribute>
            <shortName>idNameOfField1</shortName>
            <value>VALUE</value>
        </attribute>
        <attribute>
            <shortName>idNameOfField2</shortName>
            <value>VALUE</value>
        </attribute>
        ...
    </semanticGroup>
    ...
</tr>
<tr techId="id2" techName="name2">...</tr>
...
</data>

```

**response(failure):** *Not enough rights*

Listing 25: Response of the search.php script if the user is not allowed to read data.

```

<?xml version="1.0" ?>
<error reason="access denied">
    <span xmlns="http://www.w3.org/1999/xhtml">
        <h1>ACCESS DENIED</h1><h2>You have no rights to access this
            functionality</h2>
        <h2>You may log in first...</h2>
    </span>
</error>

```

<h2>You may log in first...</h2> is only shown if no user is logged in.  
 If the user is not allowed to use the search functionality, but s/he is allowed to view the data, always all existing entries are returned.

*The sys\_semGroups argument contains a name that is not a name of a semantic group defined in the current table structure XML file specified by the URL parameter cfg-pref:*

Listing 26: Response of the search.php script if the client asks to access a table the script is not allowed to.

```

<?xml version="1.0" ?>
<error reason="critical error">
    <span xmlns="http://www.w3.org/1999/xhtml">
        <h1>CRITICAL ERROR</h1>
    </span>
</error>

```

```

    <h2>Your client software sent an request that lead to an
      error. Please contact us!</h2>
  </span>
</error>

```

This error cannot appear if the unchanged JavaScript client is used, but only if someone tries to access database tables s/he is not allowed to.

*No POST - data*

Listing 27: Response of the search.php file if no or not enough POST-data was sent.

```

<?xml version="1.0" ?>
<error reason="no data">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ERROR</h1>
    <h2>No data specifying the request found</h2>
  </span>
</error>

```

If the user does not have the right to search, but only the “read”- right, all rows are selected from the database, but no error is displayed.

### 11.2.16 Export

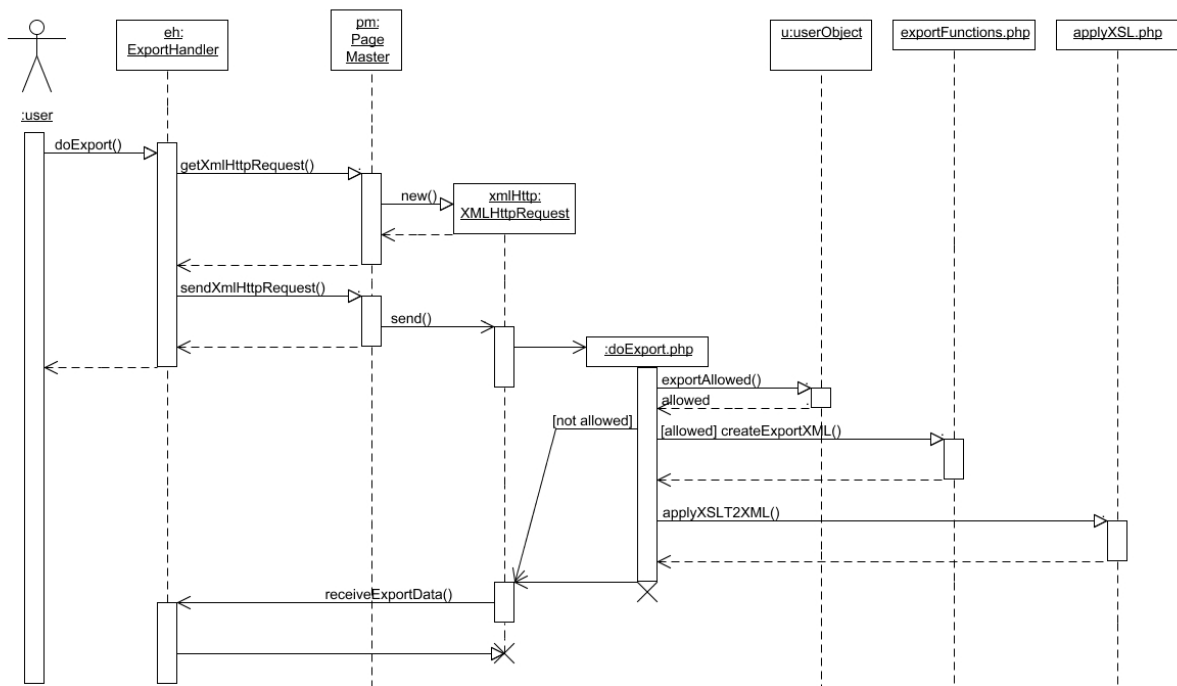


Figure 105: UML sequence diagram of the export process.

**export.php: argument/return values:**

**request:** POST - data (url encoded):

- *sys\_ids*: a semicolon separated list containing the ids of the entries that shall be in the export output.
- *sys\_semGroups*: contains a semicolon separated list containing the ids of the semantic groups (= names of the database tables) from which data shall be loaded from/attributes shall be displayed.
- *[semGroup]*: For each entry in the list of *sys\_semGroups*, an argument having the name of the database table must exist, that contains a semicolon separated list of the fields that shall be selected from this database table.
- *sys\_semGroupsV*: contains a semicolon separated list of ids of the semantic groups (= names of the database tables) from which attribute groups shall be displayed.
- *[semGroupV]\_V*: For each entry in the *sys\_semGroupsV* list, there must be an argument with the value of the entry as name, followed by “\_V”. These arguments must contain a semicolon separated list of the names of attribute groups belonging to the corresponding semantic groups that shall be displayed.
- *sys\_stylesheet*: contains the name of the stylesheet that shall be used for the transformation (without the .xsl ending)

**response(success):** The output of the XSLT transformation implemented by the XSLT stylesheet defined by the *sys\_stylesheet* argument.

**response(failure):** *Not enough rights*

Listing 28: Response of the doExport.php script if the user is either not allowed to use the export functionality or to view the data.

```
<?xml version="1.0" ?>
<error reason="access denied">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ACCESS DENIED</h1>
    <h2>You have no rights to access this functionality</h2>
    <h2>You may log in first...</h2>
  </span>
</error>
```

*<h2>You may log in first...</h2>* is only shown if no user is logged in.

*The sys\_semGroups argument contains a name that is not a name of a semantic group defined in the current table structure XML file specified by the URL parameter cfg-pref:*

Listing 29: Response of the doExport.php script if a database table shall be accessed the script is not allowed to.

```
<?xml version="1.0" ?>
<error reason="critical error">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>CRITICAL ERROR</h1>
    <h2>Your client software sent an request that lead to an
      error. Please contact us!</h2>
```

```
</span>
</error>
```

This error cannot appear if the unchanged JavaScript client is used, but only if someone tries to access database tables s/he is not allowed to.

*No POST - data:*

Listing 30: Response of the doExport.php script if no or not enough POST data was sent.

```
<?xml version="1.0" ?>
<error reason="no data">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ERROR</h1>
    <h2>No data specifying the request found</h2>
  </span>
</error>
```

*The defined stylesheet can't be found:*

Listing 31: Response of the doExport.php script if the defined stylesheet can't be found.

```
<?xml version="1.0" ?>
<error reason="no such stylesheet">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ERROR</h1>
    <h2>There is no stylesheet with this name</h2>
  </span>
</error>
```

### 11.2.17 Screenshots

The sequence diagrams for the different actions that can be performed with screenshots (add, delete) are shown in “Add screenshots” and “Delete screenshots”. We now explain the argument/return values of the `manageScreenshots.php` file.

**manageScreenshots.php: argument/return values:**

**request:** the following arguments must be given to ...

**... insert a new screenshot:**

POST - data (URL encoded):

- *sys\_insertScreen*: must contain the string “save”
- *sys\_name*: a string that shall be used as name for the screenshot entry
- *sys\_description*: a text with further information about the screenshot
- *sys\_techId*: the id of the technique the screenshot belongs to

The image must be accessible on the server using `$FILE["sys_screen"]`.

**... alter an existing screenshot:**

POST - data (URL encoded):

- *sys\_insertScreen*: must contain the string “edit”

- *sys\_name*: a string that shall be used as name for the screenshot entry
- *sys\_description*: a text with further information about the screenshot
- *sys\_techId*: the id of the technique the screenshot belongs to
- *sys\_number*: the number of the screenshot that shall be altered.

The image must be accessible on the server using \$\_FILE["sys.screen"] (if not defined, the old image is kept).

**... delete one specific screenshot:**

POST - data (URL encoded):

- *sys\_action*: must contain the string "deleteOne".
- *sys\_techId*: the id of the technique the screenshot belongs to
- *sys\_number*: the number of the screenshot that shall be deleted.

**... load all screenshots belonging to a technique for the edit page:**

POST - data (URL encoded):

- *sys\_action*: must contain the string "loadShort"
- *sys\_id*: must contain the id of the technique for which information about all screenshots shall be loaded

**... load all screenshots belonging to a technique for the view page:**

POST - data (url encoded):

- *sys\_action*: must contain the string "loadMedium"
- *sys\_id*: must contain the id of the technique for which information about all screenshots shall be loaded

**response(success):** *insert or alter a screenshot:*

These actions are called from the popup window where the data for the screenshot are defined by the user. They therefore return an XHTML page that calls the window.close() method of the popup as soon as the page is completely loaded.

*delete one or all screenshots:*

Listing 32: Response of the manageScreenshots.php script if a screenshot has been successfully deleted.

```
<?xml version="1.0" ?>
<success></success>
```

*load all screenshots belonging to a technique for the edit page:*

Listing 33: Response of the manageScreenshots.php script containing all screenshots belonging to the specified data entry on the edit page.

```
<?xml version="1.0" ?>
<data xmlns="http://www.w3.org/1999/xhtml">
```

```

<option id="screensOpt_"[numberOfScreenshot]"_"[idOfTechnuqie]
      value=[numberOfScreenshot] thumb=[pathToTheImage]">
      nameOfTheScreenshot</option>
.... (for each screenshot of this technique)
</data>

```

*load all screenshots belonging to a technique for the view page:*

Listing 34: Response of the manageScreenshot.php script containing all screenshots belonging to the specified entry on the view page.

```

<?xml version="1.0" ?>
<data xmlns="http://www.w3.org/1999/xhtml">
  <option id="screensOpt_"[numberOfScreenshot]"_"[idOfTechnuqie]
        value=[numberOfScreenshot] thumb=[nameOfTheImage]>
        nameOfTheScreenshot</option>
  <description id="screensOpt_"[numberOfScreenshot]"_"[
        idOfTechnuqie] value=[numberOfScreenshot]>'text describing
        the screenshot</description>
  .... (for each screenshot of this technique)
</data>

```

**response(failure):** *insert or alter a screenshot:*

These actions are called from the popup window where the data for the screenshot are defined by the user. They therefore return an XHTML page that contains an error message, telling the user what has gone wrong.

*delete one or all screenshots, load all screenshots belonging to a technique for the edit - or view page:*  
*not enough rights:*

Listing 35: Response of the manageScreenshot.php script if the user is not allowed to delete data.

```

<?xml version="1.0" ?>
<error reason="access denied">
  <span xmlns="http://www.w3.org/1999/xhtml">
    <h1>ACCESS DENIED</h1>
    <h2>You have no rights to access this functionality</h2>
    <h2>You may log in first...</h2>
  </span>
</error>
<h2>You may log in first...</h2> only shown if no user is logged in.

```

## 11.3 To-do's and Known Bugs

### 11.3.1 Known Bugs

- DQT is developed only for the Firefox browser (and partly Internet Explorer 6, but not with version 7).
- If using Internet Explorer 6, some display errors occur.

- The text field of integer type cells is not focused after an illegal value has been inserted by a user to the field, although the `focus()` method of the text field is called.
- If no column is active for export, nevertheless empty attribute unit header rows are exported. No rows should be exported if there are no columns selected.
- A problem occurs if a user logs out while data is loading and because of the logout loses the right to see these data. Then the element the data is loaded into (the table for example) is simply destroyed. Because of this, the loading interrupts with an error, because the element the data shall be written to is missing. To explicitly stop loading the data and only then removing the element would be preferred.

### 11.3.2 To-do's and Further Extensions

- “int” type: At the moment, if an invalid integer value is entered by a user and the first action done afterwards is to click the “save” or “search” button, saving or searching starts. It is also an error message prompted that informs about the invalid error, but the save/search actions starts too. It would be better if this would not happen, but only the error message is shown.
- “md5” type: Creating an own type object (with a special user interface) for the “md5” type. This object should allow to enter already hashed values as well as values that shall become hashed before they are stored in the database.
- A class hierarchy for data types should be introduced. It should make developing of new datatypes easier.
- “set” type: There's a problem in the user interface for editing the value of the “set” type. If the selection box is active when the `[+]` button is pressed, Firefox does not automatically update the height of the table cell. In this version of DQT a workaround is implemented, but a more sophisticated solution would be preferred.
- For some errors occurring within an `XMLHttpRequest`, the value of the field storing a reference to the corresponding request object cannot be set to `false`. (Because the program execution is aborted by this error.) Because of this, this request object cannot be used any more until DQT is restarted/ reloaded. A timer that resets the value of this field would solve this problem.
- The ordering sequence used by the `ManageColumn` object to sort its “cell” array differs from the sequence used by the `Column` object to sort its “cell” array. (This order is the same as the one used by the `TableCluster` for its “rows” array.) The use of the same ordering sequence everywhere would be preferred.
- At the moment, a lot of user interface elements is not really removed from the screen if the user loses the right to access them, but only their display property is set to `"none"`. It would be probably better to really remove these elements.
- The definitions for which user rights are necessary to access specific areas are hardcoded in all the `xyzAllowed()` methods of the user objects. Also the dependencies of access rights to the `cfg.pref` value are hardcoded in these functions. A possibility to declarative define these dependencies and rights is preferred.

- “shortName” and “idName” are both used to describe the identifier names of objects. To use a unique naming, only “idName” should be used.
- As explained in section “Documentation of Design Decisions”, the request data sent to the server are not XML formatted but encoded as POST data. It should be discussed whether using an XML format for the request data is to be preferred.
- “urllist”-type: At the moment only http is supported as protocol for URLs. Other protocols should be possible too.
- In the current version, only for the cells of semantic groups, column groups and columns additional information is shown if the mouse is over the cell. This could also be done for the cells of attribute units.
- When checking/unchecking a checkbox that controls some functionality (like the visibility of columns), this functionality often freezes the browser. The browser therefore cannot respond to the action of the user and update the display of the checkbox. A solution could be to not call the function directly, but to use a timer that waits 10ms before the function is called.
- The rendering of bigger tables takes all browsers quite a long time. It should be discussed whether other useful ways of displaying the data exist.

## 11.4 Documentation of Design-Decisions

### Not using a framework/library

Before the implementation of DQT, the author did little researches for a framework or a JavaScript library that could help for implementation. Especially a library providing possibilities for the table management was looked for. No open source libraries were found that could help with respect to the table management. In general, the use of the frameworks available at this time was not regarded to be useful, as their functionality was very limited. The frameworks have been improved a lot since the researches, so it may would make sense now to adopt DQT to a framework.

### Not creating a pure Ajax application

DQT is no pure Ajax application. The navigation between the edit and the view is not performed using the Ajax approach, but by simply loading a new page. This could be seen as a break in the user experience.

The main reason for this was to clearly and strictly separate these parts of the application. The user shall be aware of the fact that s/he’s entering a completely different part of the application when navigating from the view to the edit part or vice versa. So this separation was not ought to be a break in the look and feel, but a good mix between the traditionally and the Ajax approach. Within a site, Ajax is used to not interrupt the work of the user only to load new data. When changing the site, the work of the user must be interrupted, as the complete user interface needs to be rebuild. The use of Ajax would not bring any benefit in this case.

Additionally, the code of DQT would be much more complex if all these functionalities would have been implemented in one site.



## Not using XML formatted requests

Although the name of DQT contains “XML”, only the server output is XML formatted. The requests sent by the client are formatted as “key=value” pairs, just like the POST-data received by a server in traditional webapplications. The main reason for this was, that it should be easy to create a traditional front end for DQT that could be used by persons who use browsers not supported by DQT. The chosen request format would allow to use the same server scripts (with only small changes for the output) for this front end. Another reason was that php provides a simple way of accessing POST variables formatted like this.

## 12 Conclusion

Within a 12.5 ETCS practical course, the “Dynamic XML Query Table Editor” (DQT), a new web application for the management and presentation of collected data, was developed. This thesis contains its documentation.

DQT was implemented using the new “Ajax” [Garret,2005a] approach, which allows to create web applications that perform like desktop applications. DQT benefits from the feature of sending data to the server (or receive data from the server) without the need to completely load a new page. This is not only an advantage in data editing, but also improves the handling of the search and export functionality.

Although the extensive use of JavaScript facilitates the handling of DQT for the user, it complicated the software development. Especially debugging of JavaScript within the browsers was very unhandy. A practical debugging tool that overcomes this problem, but was only published recently, is version 0.3 of the Firefox extension “FireBug” [Hewitt,2006]. Other Firefox extensions [Mozilla,2003; Pederick,2006; Madden,2006] were also helpful during development of DQT.

Another problem which occurred during implementation was that although JavaScript implementations of the different browsers behave almost equally, still some differences exist. Nevertheless the author considers the use of the “Ajax” approach made DQT much more comfortable for the end user than a “traditional” web application would have been.

This thesis describes background information of DQT, provides an installation and configuration guide, a user manual for DQT end users and a high level documentation of DQTs implementation.

Additional low level (source code) documentation of DQT can be found in the complete source files, located in the */src* directory of the DQT package.

Javadoc like comments for all classes are available, but only within the source files. This is because no free tool was found to create HTML pages out of the comments in the JavaScript files. For example the most popular free tool for this purpose, JsDoc [Reid et al.,2006], is not capable of handling the large amount of comments.

A list of possible further improvements and extensions is also given. Additional extensions could be an import function for data or the possibility to use other permanent storages than the MySQL database.

Systematically tests have been performed successfully. Section 11.3.1 contains a list of known bugs and problems.

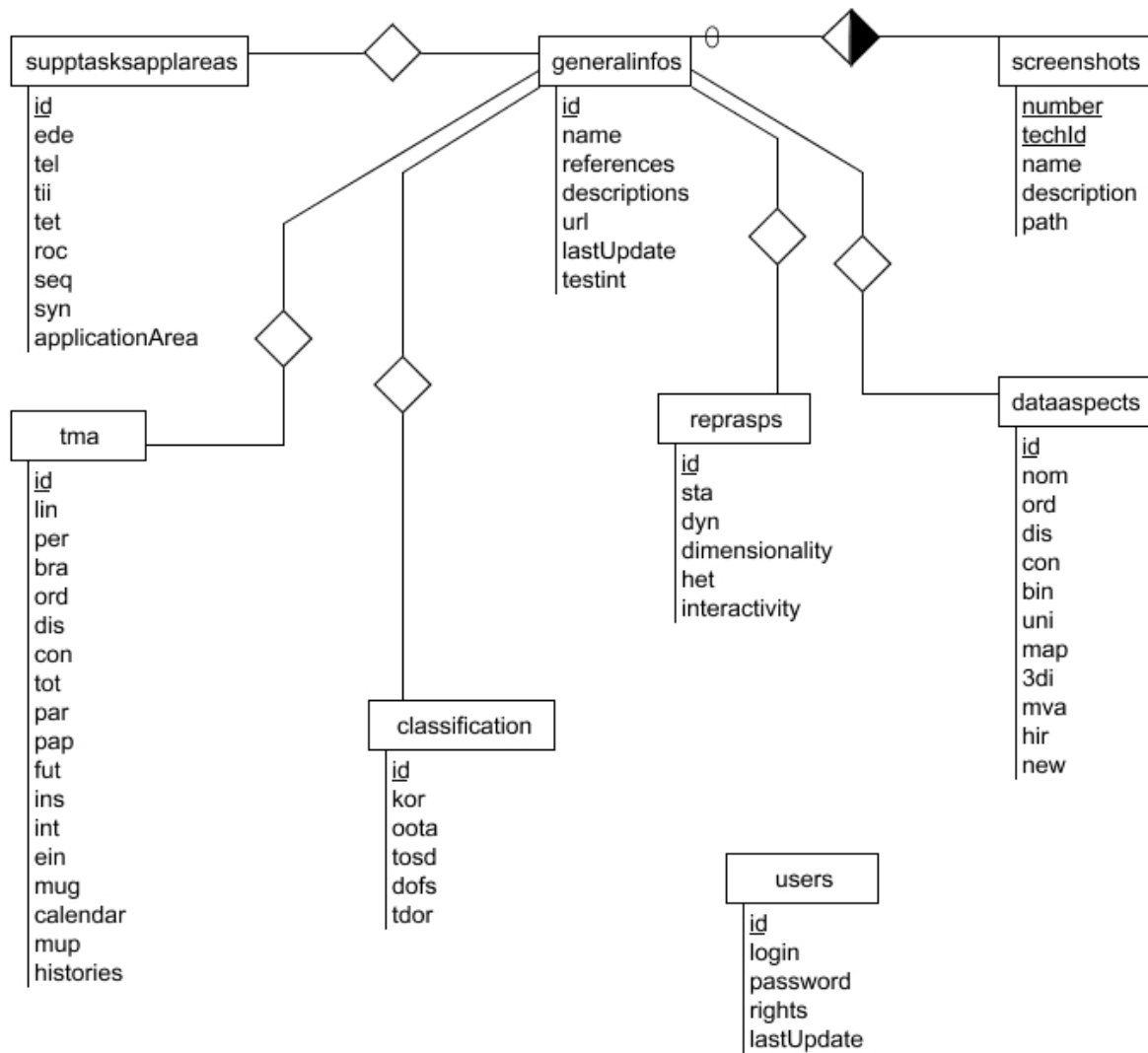
## References

- [Garret,2005a] Jesse J. Garrett, Ajax: A New Approach to Web Applications. Adaptivepath. Created at: February 18, 2005. Retrieved at: November 27, 2005. <http://www.adaptivepath.com/publications/essays/archives/000385.php>
- [Aigner,2006] W. Aigner, Visualization of Time and Time-Oriented Information: Challenges and Conceptual Design, PhD Thesis, Institute of Software Technology & Interactive Systems, TU Wien, 2006.
- [Skritek,2005] S. Skritek, Does it make sense using Ajax for a Dynamic XML Table-Editor? Term paper, TU Wien, 2005.
- [Rivest,1992] R. Rivest, Request for Comments: 1321 - The MD5 Message-Digest Algorithm. IETF. Created at: April 1992. Retrieved at: July 27, 2006. <http://www.ietf.org/rfc/rfc1321.txt>
- [PHPMyAdmin,2006] The phpMyAdmin project, phpMyAdmin. Retrieved at: July 27, 2006. <http://www.phpmyadmin.net/>
- [XAMPP,2006] Apache Friends, XAMPP. Retrieved at: July 23, 2006. <http://www.apachefriends.org/en/xampp.html>
- [Hewitt,2006] Joe Hewitt, FireBug. Retrieved at: July 27, 2006. <http://www.joehewitt.com/software/firebug/>
- [Mozilla,2003] Mozilla, DOM Inspector. Retrieved at: July 27, 2006. <http://www.mozilla.org/projects/inspector/>
- [Pederick,2006] Chris Pederick, Web Developer Extension. Retrieved at: July 27, 2006. <http://chrispederick.com/work/webdeveloper/>
- [Madden,2006] Jennifer Madden, View Source Chart (Firefox Extension). Retrieved at: July 27, 2006. <http://jennifermadden.com/scripts/ViewRenderedSource.html>
- [Reid et al.,2006] Gabriel Reid, Michael Mathews, Scott Connelly. JsDoc. Retrieved at: July 27, 2006. <http://jsdoc.sourceforge.net/>

## 13 Appendix

### 13.1 Database Structure of “Techniques for Visualization of Temporal Data”

#### 13.1.1 EER Diagram:



### 13.1.2 Database Description:

#### General Information

name	values/type	constraints	comments
name	VARCHAR(200)	unique, NOT NULL	the value of this field defines the name of an entry FULLTEXT index
id	INT(10)	unique, NOT NULL	primary key, auto_increment
references	LONGTEXT	NOT NULL	FULLTEXT index
description	LONGTEXT	NOT NULL	FULLTEXT index
url	LONGTEXT	NOT NULL	FULLTEXT index
lastUpdate	TIMESTAMP	NOT NULL	ON UPDATE CURRENT_TIMESTAMP, default: CURRENT_TIMESTAMP

Table 3: The fields of the database table “generalinfos” (abbr. for “general information”).

## Time Model Aspects

name	values/type	constraints	comments
id	INT(10)	unique, NOT NULL	maps to generalinfos.id (foreign key not possible)
lin	ENUM('f','p','n')	NOT NULL	default: 'n'; abbr. for "linear"
per	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "periodic"
bra	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "branching"
ord	ENUM('f','p','n')	NOT NULL	default: 'n'; abbr. for "ordinal"
dis	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "discrete"
con	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "continous"
tot	ENUM('f','p','n')	NOT NULL	default: 'n'; abbr. for "total"
par	ENUM('f','p','n')	NOT NULL	default: 'n'; abbr. for "partial"
pap	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "past/present"
fut	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "future"
ins	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "instant"
int	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "interval"
ein	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "explicit indeterminacy"
mug	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "mutiple granularities"
calendar	ENUM("", 'none', 'not fixed', 'Gregorian')	NOT NULL	default: ""
mup	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "multiple perspectives"
histories	LONGTEXT	NOT NULL	FULLTEXT index abbr. for "perspectives - histories"

Table 4: Fields of the "tma" database table (abbr. for "ime model aspects").

## Data Aspects

name	values/type	constraints	comments
id	INT(10)	unique, NOT NULL	maps to generalinfos.id (foreign key not possible)
nom	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "nominal"
ord	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "ordinal"
dis	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "discrete"
con	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "continuous"
bin	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "binary"
uni	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "univariate"
map	ENUM('f','p','n')	NOT NULL	default: 'n'
3di	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "3-dimensional"
mva	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "multivariate"
hir	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "hierarchical"
new	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "network"

Table 5: Fields of the database table “dataaspects” (abbr. for Data Aspects).

## Representational Aspects

name	values/type	constraints	comments
id	INT(10)	unique, NOT NULL	maps to generalinfos.id (foreign key not possible)
sta	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "static"
dyn	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "dynamic"
dimensionality	SET('1D', '1.5D', '2D', '2.5D', '3D')		default: NULL
het	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "heterogenous"
interactivity	LONGTEXT	NOT NULL	FULLTEXT index

Table 6: Fields of the database table “reprasps” (abbr. for “Representational Aspects”).

## Supported Tasks and Application Areas

name	values/type	constraints	comments
id	INT(10)	unique, NOT NULL	maps to generalinfos.id (foreign key not possible)
ede	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "existence of a data element"
tel	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "temporal location"
tii	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "temporal interval"
tet	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "temporal texture"
roc	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "rate of change"
seq	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "sequence"
syn	ENUM('f','p','n')	NOT NULL	default: 'n' abbr. for "synchronization"
applicationArea	LONGTEXT	NOT NULL	FULLTEXT index

Table 7: Fields of the database table “supptasksapplareas” (abbr. for “supported tasks and application areas”).



## Classification

name	values/type	constraints	comments
id	INT(10)	unique, NOT NULL	maps to generalinfos.id (foreign key not possible)
kor	ENUM("", 'data', 'information', 'time')	NOT NULL	default:" abbr. for "kind of representation"
oota	ENUM("", 'linear', 'periodic', 'branching')	NOT NULL	default:" abbr. for "organization of time axis"
tosd	ENUM("", 'plot_single', 'plot_cartesian', 'plot_multiple', 'plot_cyclic', 'map', 'diagram', 'chart_pixel-based', 'chart_glyph-based', 'chart_timeline-based', 'chart_calendar-based', 'chart_clock-based', 'chart_other')	NOT NULL	default:" abbr. for "type of symbolic display"
dofs	ENUM("", '2D', '3D')	NOT NULL	default:" abbr. for "dimension of represent. space"
tdor	ENUM("", 'static', 'dynamic')	NOT NULL	default:" abbr. for "time depend. of represent."

Table 8: Fields of the database table "classification".

## Screenshots

name	values/type	constraints	comments
number	INT(10)	NOT NULL	PRIMARY KEY (with techId)
techId	INT(10)	NOT NULL	PRIMARY KEY (with number)
name	VARCHAR(200)	NOT NULL	FULLTEXT index
path	VARCHAR(200)	NOT NULL	FULLTEXT index
description	LONGTEXT	NOT NULL	FULLTEXT index

Table 9: Fields of the database table "screenshots"

## Users

name	values/type	constraints	comments
id	INT(10)	unique, NOT NULL	auto_increment, PRIMARY KEY
login	VARCHAR(100)	unique, NOT NULL	
rights	SET('insert', 'delete', 'edit', 'export', 'user', 'search', 'read')	NOT NULL	
password	VARCHAR(100)	NOT NULL	
lastUpdate	TIMESTAMP	NOT NULL	ON UPDATE CURRENT_TIMESTAMP, default: CURRENT_TIMESTAMP

Table 10: Fields of the database table “users”