Easy Database Management in C++ Applications by Using DatabaseExplorer Tool

Peter Janků, Michal Bližňák

Abstracts—One of the most important task in an application development process is a database management and data manipulation. Nearly every programming toolkit includes tools, constructs or components for manipulation with database content and structure. DatabaseExplorer is an open-source cross-platform software tool for C++ programming language and wxWidgets toolkit which makes database structure management and data manipulation easy. It provides simple and intuitive GUI interface for handling of multiple different database engines and stored data. The application can be used as a stand-alone tool or as a plugin for well known CodeLite IDE. This paper deals with the application description and reveals its internal structure, used technologies and provides simple use case scenario.

Keywords—Database, ERD, wxWidgets, C++, explorer, table, view, SQL, script

I. Introduction

Ever since computers were invented, developers have been trying to create devices and software for faster, easier and more efficient data manipulation. First devices worked only with inputs and outputs and they didn't take care about data storing. But needs of data storing increased fast over the time. Together with these tendencies increased needs of using data solid structure too. Nowadays, data archiving and manipulation is one of the most important tasks in IT.

If we look at data storing process from an ordinary user's point of view and we can omit data storage technological aspects we reveal that many things are getting easier. The current trends bring up technologies, programming languages and visual formalisms which are very clear and intuitive to use and understand such are various database engines, SQL language (and its dialects) [3] or ERD diagrams.[13]

There are lot of open source and commercial applications able to manipulate data stored in databases, but they often lack of some basic needed functionality like:

- **platform independence** – application should be able to run on multiple different operating systems such are MS Windows, MacOS and Linux,
- **support for multiple database engines** – application can manipulate data and database structure under multiple different database engines[1],
- **ERD support** – user can use ER diagrams for formal definition of database structure and for forward and reverse engineering.

[1]In this article "database engine" will be regarded as a complete system determined for handling data. The system includes database server, storage engine and data base management system (DBMS).[1]

DatabaseExplorer tool (shortly DBE) presented in this paper was developed to fill the market gape, i.e. to offer all functionality mentioned above in one easy to use application with friendly user interface. Thanks to its concept and used technologies it allows users to handle data stored in several different database systems (MySQL, PostgreSQL and SQLite) and provides an intuitive way how to work with the database structure by using ERDs.

II. Used Technologies

As mentioned in the previous paragraphs, DatabaseExplorer tool is written in the C++ language due to its performance capabilities and due to a fact that the application was intended to be published also as a plug-in available for CodeLite IDE [10] which is written in C++ as well. The cross-platform library **wxWidgets** [12] was used for implementation of the application's GUI and encapsulation of other needed platform-dependent functionality like access to database engines, file system, etc. Thanks to the library the application provides fully native GUI interface for all supported platforms (demonstration of native GUI is shown in Fig. 1) and it runs nearly as fast as applications written by using native platform-dependent APIs.

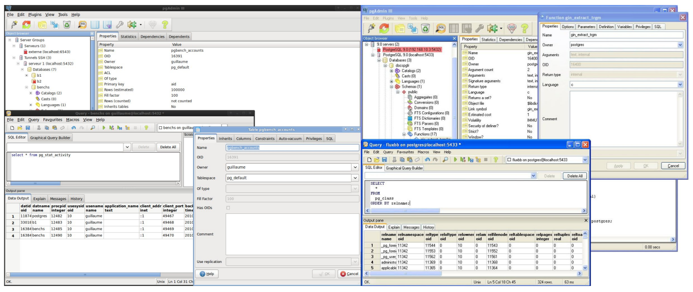


Fig. 1 Demonstration of native GUI provided by wxWidgets (GTK-based GUI on the left, WIN32API-based GUI on the right)

A. 3rd-party Application Components

In addition to standard wxWidgets library components also several 3rd-party add-ons have been used.

DatabaseLayer (shortly DBL) is first of those components. It provides generic interface for manipulation with various databases by sending an SQL scripts to the database engine and receiving the results. So far SQLite3, PostgreSQL, MySQL, Firebird, and ODBC database backends are supported [9]. DBL library represents a bottom layer of DatabaseExplorer application[2].

[2]Full application structure is described in chapter III. of this article.

The **wxShapeFramework** add-on (shortly wxSF) is a software library/framework based on wxWidgets which allows easy development of software applications manipulating with graphical objects (shapes) like various CASE tools, technological processes modeling tools, etc [8]. The add-on is used in DatabaseExplorer for creation and manipulation with ER diagrams.

The **wxXmlSerializer** add-on was (shortly wxXS) used as a persistent data container for internal database structure snapshot. The library offers a functionality needed for creation of persistent hierarchical data containers which can be serialized/deserialized to/from an XML structure.[7].

Also wxSF itself uses the persistent data container provided by wxXS. As mentioned wxXS allows users to easily serialize and deserialize hierarchically arranged class instances and their data members to an XML structure. The XML content can be stored to a disk file or to another output stream supported by wxWidgets. This functionality is used for saving and loading of diagrams as well as a base for the clipboard and undo/redo operations provided by the wxSF [8]. Relation between wxSF and wxXS libraries is explained in Fig. 2

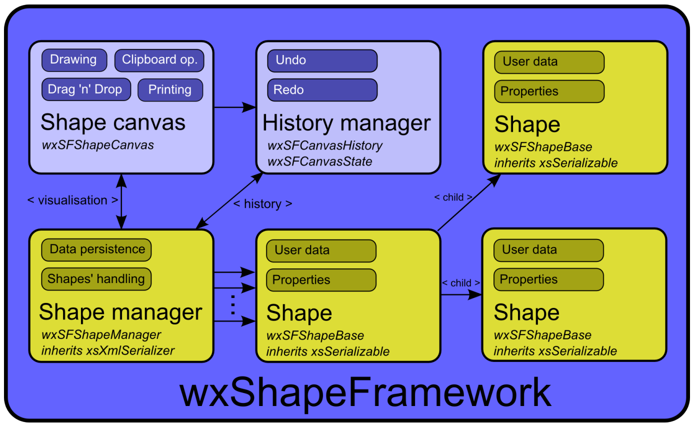


Fig. 2 Logical structure of wxSF library

B. *Supported Database Engines*

So far two server-based databases and one file-based database are supported by current version of DBE. Thanks to an abstract database interface used in the application and DBL library also another database engines can be easily added in a future.

The first supported server-based database is **MySQL** which is well known database server mainly used for various web applications. Of course, thanks to published software connector sources the MySQL server can be easily accessed also from any desktop applications written in supported programming language. MySQL from its begins was designed as a lightweight database server which offers just basic features and is optimized to transaction speed. Currently also set of enhanced database functions including views, stored procedures, triggers, etc are provided by the database engine.[4]

An interesting aspect of the MySQL database server is it's internal structure. The server software is divide into a three basic layers. The top layer includes a "connection pool" which can manage a database connections, the middle layer encapsulates parser, optimizer and other components needed for server functionality and the bottom layer provides pluggable storage engines. These engines allow users to select compromise between engine speed, data size and available database features.[5]

The second supported server-based database is **PostgreSQL**. It is widespread, open-source, cross-platform power-full database server which can be installed on various operating systems. PostgreSQL have few interesting features. The first one is that the server implements SQL language strictly corresponding with ANSI-SQL:2008 standard. Of course, it should be a standard approach for all database servers' developers, but if you look at the others, you can see a lot of small modifications (dialects) of standardized SQL language.

PosgreSQL provides all "standard" features like triggers, views, stored procedures, indexing, transactions, etc. Beside this functionality we can use also for example GiST indexing system, internal stored procedures which can be written in multiple languages (Java, Perl, Python, Ruby, ...) and special attribute types of stored data (geometry points, graphical object, IP addresses, ISBN/ISSN numbers and others) [6].

Finally, **SQLite** is the last database engine currently supported by DBE. It is cross-platform, small and powerfull file-based database. The main advantage of the engine is its very small memory and system footprint which makes it and ideal solution for build-in database library. SQLite can mediate benefits of transaction databases to simple applications without needs of any network connection. SQLite doesn't contain any server side part. Every data are directly written to/read from a file stored in a file system; complete database with all tables, views and triggers is stored in one database file.

III. DatabaseExplorer Architecture

DBE application's internal structure is divided into three cooperating software layers. The architecture is shown in Fig. 3.

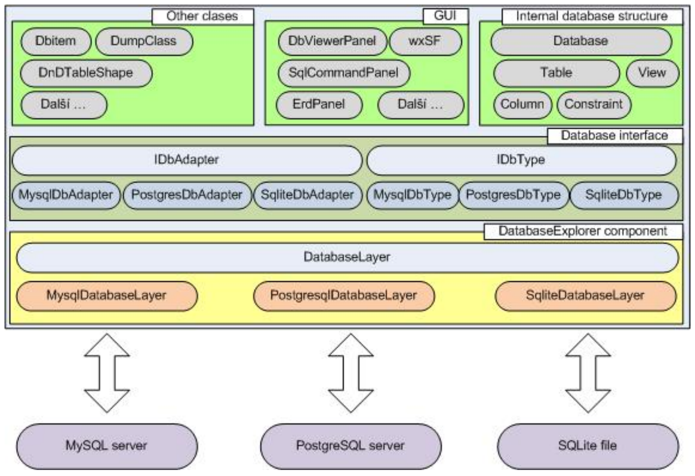


Fig. 3 DatabaseExplorer architecture

The bottom layer is presented by DBL component. DBL provides unified interface and encapsulates multiple software database connectors which can be used for connection to relevant databases. This layer is used for establishing the database connection, sending SQL command to it and for receiving requested results.

The middle layer of the architecture implements two interfaces - *IDbAdapter* and *IDbType*. *IDbType* is an abstract class which defines basic functions for manipulation with database types. This class and its implementations are used for database type unification. The second abstract class *IDbAdapter* provides all functions needed for database structure manipulation. Via its implementations the application can unify manipulation with database structure on multiple different database servers.

The top layer of DBE application contains all other classes encapsulation the rest of the application functionality. They use the abstract interfaces provided by the middle layer, so they don't care about connected database types. There are also GUI-related classes, classes for source code generation, and set of classes which can create internal database structure snapshot.

A. *Accessing Different Database Engines*

Every supported database system has its own database catalogue which stores an information about database structure. In fact it means that there are few different methods for reading this structure an there is no universal interface defined in software connector sources for accessing the database in a standardized way. This is problem for an application which wants to bring up universal GUI for changing database structure on multiple different database servers. Due to this reasons DBE uses only SQL commands for reading and writing database structure and data. These commands are specific for every database server and thus their has to be modified in accordance to the target database engine. This task is done by DBE which converts internally used universal data types and database objects into target-specific ones.

B. *DatabaseExplorer Distribution Types*

As mentioned above the DBE is implemented in two different ways. It can be obtained as a standard stand-alone application which can be installed and used separately [11]. It is also implemented as a plug-in for CodeLite IDE [10]. Both implementations share the most of the application's source code and functionality and differ only in minor GUI implementation aspects. Note that all screenshots dealing with DBE's GUI presented in this paper are taken from the CodeLite's DBE plugin.

IV. Basic Application Usage

As shown in Fig. 4, the DBE GUI consists of two basic parts; a notebook which manages tabs containing every opened SQL editor (Fig. 5)/ERD editor (Fig. 6) and *DBE main panel* including tree control allowing user to access all opened databases and their objects (such are database tables and views) and active ERD's thumbnail. In this context an *"editor"* means a panel with GUI controls (SQL command editor and database table grid) allowing users to work with table's/view's data. This corresponds with windows' structure and layout used in CodeLite IDE. Thanks to this concept DBE can be easily integrated with CodeLite IDE as mentioned in chapter III.B..

Upper part of *DBE main panel* contains toolbar with buttons used for opening/closing of database connections, refreshing the database tree and for creation of empty ERD. Tree control placed under the toolbar shows tree structure representing real database structure of connected database. User can navigate through the structure and open suitable editor for selected database object. Editor can be opened by left mouse button double-click onto relevant tree item or by using the item's context menu. The context

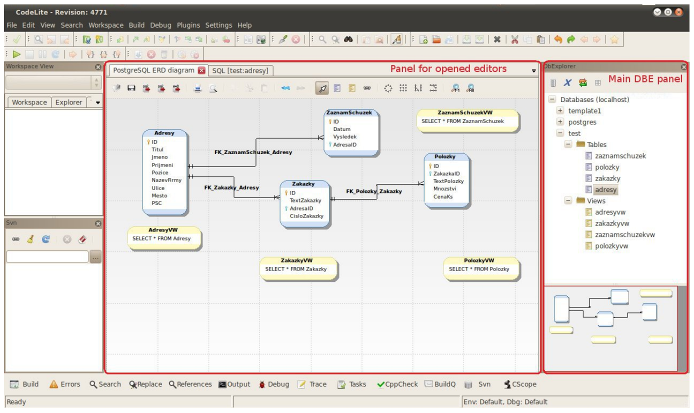
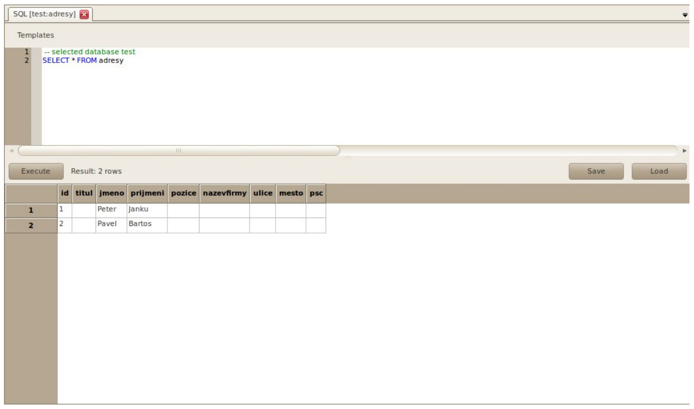


Fig. 4 CodeLite IDE window with DatabaseExplorer plugin

menu can be also used for import/export functions and for creation of new database as well. Note than the tree structure shows data in off-line mode by using internal database snapshot so a user have to perform *Refresh* for sync this structure every-time the content of source database changes.

In the bottom part of main panel is shown thumbnail of currently selected ER diagram. If no diagram is select then the thumbnail is empty.

As mentioned above there are two editor types in DBE. The first one represents an SQL command panel shown in Fig. 5. It consists of a multi-line text area where an SQL query can be written to and of a database table grid showing result of previously performed SQL query. These queries can be simple send to the database via button placed on the bottom-left side of the query editor and user can view their results immediately. SQL queries created in this editor can be easily saved into text file with standard extension "*.sql". SQL files can be also imported into the editor as well. Buttons providing this functions are places on the bottom-right side of the SQL editor [3]. Moreover, there is a menu with SQL query examples placed in the top of the editor which allows users to add an example script into the text area which can help him to follow the correct SQL syntax.



```
-- selected database test
SELECT * FROM adresy
```

Result: 2 rows

	id	titul	jmeno	prijmeni	pozice	nazevfirmy	ulice	mesto	psc
1	1		Peter	Janku					
2	2		Pavel	Bartos					

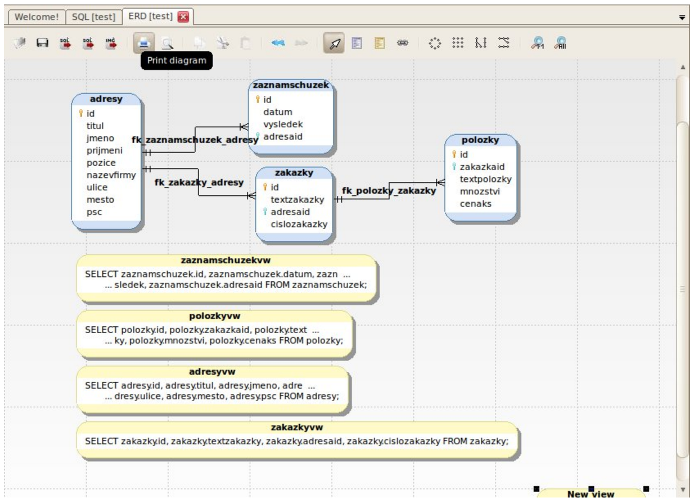
Fig. 5 SQL command editor

The second editor type shown in Fig. 6 is used for manipulation with ERDs. ER diagrams in DBE can be used for database

[3] "*.sql" file format stores SQL queries as a plain text so it does not matter in which editor the files were created

structure definition or modification as well as for graphical representation of existing database structure. Thanks to used graphical backend (wxSF) the ERD displayed in the canvas placed in the editor can be interactively created or modified. The editor contains also toolbar located in the top of the editor panel offering lot of design tools such are shortcuts for printing functions, undo/redo, clipboard operations and auto-layouting features provided by wxSF library. Also three types of database objects can be put onto the canvas - *database table*, *database view* and connection lines representing *foreign keys*.

ERD Table [2] object represents one database table defined in modified database structure. It associates all table columns (attributes) with data types and other important parameters and all table constraints including primary and foreign keys. Database table properties (i.e. properties of data columns and primary/-foreign keys) can be set or modified via dialog which can be opened by left mouse button double-click onto the table object or via table object's context menu.



```
SELECT zaznamschuzek.id, zaznamschuzek.datum, zazn ...
     ... sledek, zaznamschuzek.adresaid FROM zaznamschuzek;

SELECT polozky.id, polozky.zakazkaid, polozky.text ...
     ... ky, polozky.mnozstvi, polozky.cenaks FROM polozky;

SELECT adresy.id, adresy.titul, adresy.jmeno, adre ...
     ... dresy.ulice, adresy.mesto, adresy.psc FROM adresy;

SELECT zakazky.id, zakazky.textzakazky, zakazky.adresaid, zakazky.cislozakazky FROM zakazky;
```

Fig. 6 ERD editor

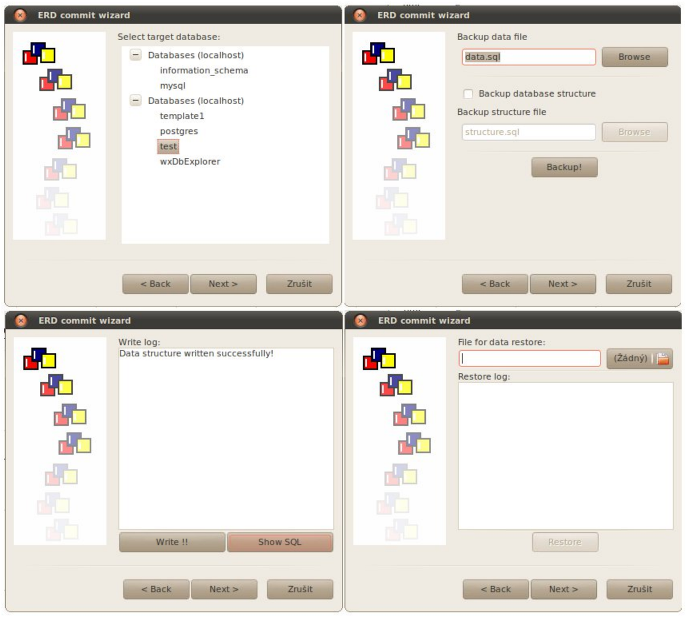
ERD View [2] object represents one real data view defined in database structure. Database view is clearly defined by name and SQL *SELECT* [3] statement. These parameters can be set via simple dialog which can be opened by left mouse button double-click onto the view object or via view object's context menu.

Foreign key connection line [2] placed between two tables represents one real constraint defined in one of the connected tables. The connection line can be created by mouse dragging when appropriate tool is selected in the editor's toolbar. In this case, a special dialog for constraint parameters definition is displayed. The second way how define the constraint is via database table properties dialog as mentioned above. This is also the only way how an existing foreign key connection line can be edited.

ERD editor provides also other useful functionality including possibility to save ERD into XML file, generation of SQL scripts, etc. The features can be access via editor's toolbar or via ERD's context menu.

In addition to possibility of manual ERD definition the diagrams can be generated also automatically from existing database structure. This function is available from database's context menu displayed in main DBE panel.

User can also add existing ERD objects to displayed ERD diagram by simple dragging the database objects from database structure tree into the opened editor canvas. Thanks to this features user don't have to redefine all tables again and again if he wants to modify existing tables or databases. Note that imported database object don't have to match the target database type (i.e. source database adapter type differs from the target one - for example if the user wants to drag database table object from MySQL database to SQLite database, etc.). Thus, during the import of any database object all contained data types are converted to proper data types supported by target database automatically. Moreover, data type implicit conversion can occur also when the data are written into database.[4].



```
Data structure written successfully!
```

Fig. 7 Basic steps of ERD commit wizard

If ERD diagram is not empty the described database structure can be written into selected database server. This process is provide by simple wizard shown in Fig. 7. The wizard dialog navigates user through few steps including selection of target database, backup of its current database structure and data, write of new structure into the database server and restoration of backed-up data.

V. Generation of Database Access Classes

Another very useful feature provided by DBE which can make programmers' life much more easier is generation of source code implementing classes suitable for database tables access. This feature allows users to generate necessary source code for access to selected databases. If the function is invoked the header and source C++ files are generated for every database table and view contained in the database. The generated classes are following:

The first class name corresponds to the database object name (for example if the table's name is *Orders* then the class name is *Orders* as well). This class encapsulates data members related to

[4]For example a database view can be redefined by the target database engine to correspond with returned column count. Another case: a data type "serial" is transform into data type "integer" with special object "sequence" in PostgreSQL.

all table (or view) columns and the class instance represents one table row (database record).

The second class has name is composed of the table name and suffix "Col" (for example *OrdersCol*). This class represents collection of table rows and encapsulates data array containing instances of previously mentioned data record class - one class instance for each table row. Except the data array the generated class contains also member functions suitable for loading of data from database server into the class or for filling of a data grid GUI control.

The third class is generated for database table only (not for views). The name of the class is composed of the source table name (like in two previous cases) and suffix "Utils" (for example *OrdersUtils*). The class encapsulates additional functions suitable for addition, modification and deletion of table rows.

All generated classes use DatabaseLayer component for accessing the underlying database engines. If DBE is used as CodeLite IDE plug-in then also target project and virtual folder can be selected so after the code generating the application adds all generated classes into selected virtual folder of the target project. All mentioned classes are generated over textual template files so the generated content can be easily personalized if needed.

The common use case scenario can be as follows. User should:

1. define database structure by using ERD,
2. write/export database structure directly into the database server,
3. generate database access classes
4. and use the classes in his project like illustrated in a code listing bellow:

A. *Example*

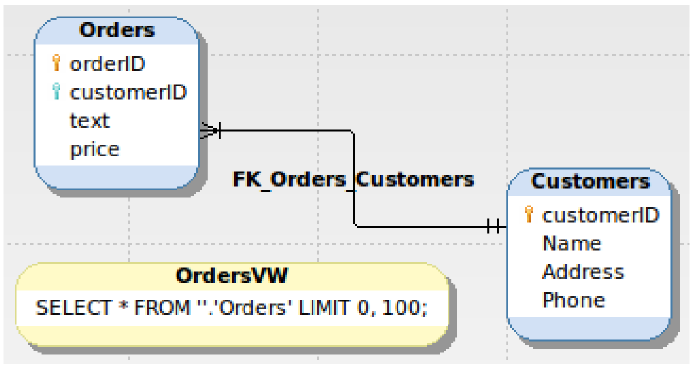


Fig. 8 Example ERD diagram

The following listings illustrate generated source code and SQL script based of ERD shown in Fig. 8.

SQL script for creation of database structure is as follows:

Listing 1 SQL script generated by DBE from ERD shown in Fig. 8:

```
-- SQL script created by DatabaseExplorer

DROP TABLE IF EXISTS 'Customers';
CREATE TABLE 'Customers' (
         `customerID` INTEGER NOT NULL,
         `Name` TEXT NOT NULL,
         `Address` TEXT NOT NULL,
         `Phone` TEXT NOT NULL,
        PRIMARY KEY ('customerID'));
-- ------------------------------------------------

DROP TABLE IF EXISTS 'Orders';
CREATE TABLE 'Orders' (
         `orderID` INTEGER NOT NULL,
         `customerID` INTEGER NOT NULL,
         `text` TEXT,
         `price` REAL NOT NULL,
        PRIMARY KEY ('orderID'),
        FOREIGN KEY('customerID')
           REFERENCES Customers('customerID'));
-- ------------------------------------------------

DROP VIEW IF EXISTS `OrdersVW`;
CREATE VIEW `OrdersVW` AS
        SELECT * FROM ''.'Orders' LIMIT 0, 100;
-- ------------------------------------------------
```

Listing 2 Header file generated by DBE from ERD shown in Fig. 8 for Orders table:

```
#ifndef ORDERS_H
#define ORDERS_H

#include <wx/wx.h>
#include "DatabaseLayer.h"

/*! \brief Class for Orders table */
class OrdersBase {
public:
  /*! \brief Constructor for loading from DB */
  OrdersBase(DatabaseResultSet* pResult);
  virtual ~OrdersBase();

  const int GetorderID() const {
    return m_orderID;
    }
  const int GetcustomerID() const {
    return m_customerID;
    }
  const wxString& Gettext() const {
    return m_text;
    }
  const double Getprice() const {
    return m_price;
    }
  /*! \brief Return Orders from db on the
      orderID base */
  static OrdersBase* GetByorderID(int orderID,
      DatabaseLayer* pDbLayer);

protected:
  int m_orderID;
  int m_customerID;
  wxString m_text;
  double m_price;
};

#include <wx/grid.h>
#include <wx/list.h>
```

```
WX_DECLARE_LIST(OrdersBase, OrdersBaseList);

/*! \brief Collection from OrdersCollectionBase
     table */
class OrdersCollectionBase {
public:
  /*! \brief conditions connetion type */
  enum CondConType
  {
    wAND = 1,
    wOR = 2
  };

  /*! \brief Constructor for loading from db */
  OrdersCollectionBase(DatabaseResultSet*
       pResult);
  virtual ~OrdersCollectionBase();
  /*! \brief Fill wxGrid from collection. */
  void Fill(wxGrid* pGrid);
  /*! \brief Get item list */
  const OrdersBaseList& GetCollection() const {
       return m_list; }
  /*! \brief Get all data from database */
  static OrdersCollectionBase* Get(
       DatabaseLayer* pDbLayer);
  /*! \brief Get data from database with WHERE
       statement */
  static OrdersCollectionBase* Get(
       DatabaseLayer* pDbLayer, wxArrayString&
       conditions,CondConType conType = wAND );

protected:
  OrdersBaseList m_list;
};

/*! \brief Utils for Orders table */
class OrdersUtilsBase {
public:
  /*! \brief Add new item into Orders table */
  static int Add( DatabaseLayer* pDbLayer
       ,int orderID
       ,int customerID
       ,const wxString& text
       ,double price
       );
  /*! \brief Edit item in Orders table */
  static int Edit(DatabaseLayer* pDbLayer
       ,int orderID
       ,int customerID
       ,const wxString& text
       ,double price
       );
  /*! \brief Delete item from Orders table */
  static int Delete(DatabaseLayer* pDbLayer
       ,int orderID
       );
};

#endif // ORDERS_H
```

VI. Step-By-Step User Guide

All described steps assume that DatabaseExplorer plug-in for CodeLite IDE is used but it is applicable also for the stand-alone version. In this tutorial we are going to create simple database containing tables for storing customer's orders, ordered items and the customer addresses. After the database definition we will create simple application written in C++ and wxWidgets which will use our new database via automatically generated classes.

A. *Definition of Database Structure via ERD Diagram*

The first step of creating of the new database application is a database structure definition. Every database usually contains tables, views, their relations and data restrictions - constraints. The easiest way how to define a database structure is by using ERDs. ERD is a simple graphical diagram with shapes which can easily describe all database parts and their relations. Thanks to the ERDs a user can simply imagine database structure with all its dependencies.

Firstly, let's create a new ERD containing data tables and views. The fist table placed into the diagram is table called *"Orders"*. The table containing list of all customer's orders will consist of several columns:

- **ID** *int auto increment not null* – unique identification of the order
- **addressID** *int not null* – ID of customer's address
- **orderDate** *datetime not null* – order's date and time

After that we can create some constraints. At the moment we will create just one constraint with name *"PK_Orders"*. This constraint is a primary key type and it is connected with *"ID"* column.

The second table in the ERD is table called *"OrderItems"* and contains items of our orders. Columns defined in this table are:

- **ID** *int auto increment not null* – unique identification of order item
- **orderID** *int not null* – ID of parent order
- **itemText** *text* – order item name
- **pcs** *int not null* – count of items
- **pricePerPcs** *float int not null* – price per an item

In this table we should create two constraints. Like in the previous case, the first constraint is a primary key and it is defined on *"ID"* column with *"PK_OrderItems"* name. The second constraint is foreign key type constraint consisting of a local column, reference table and reference column. In this case the local column will be *"orderID"*, the reference table will be *"Orders"* and the reference column will be *"ID"*. The foreign key is represented by a connection line between tables *"Orders <–> OrderItems"* and uses the primary key defined in the *"Orders"* table. This connection is *1:N* type which means that one row in table *"Orders"* can be connected with multiple rows in table *"OrderItems"*.

The third table will have name *"AddressBook"*. This table will be used for customer address and other information archivation. Columns of table *"AddressBook"* can be:

- **ID** *int auto increment not null* – unique identification of customer
- **name** *text* – customer's name
- **address** *text* – customer's address
- **phone** *text* – customer's phone number
- **email** *text* – customer's email address

It's a good idea to create a primary key for this table as well.

If we have all the tables created we have to go back to the *"Orders"* table properties and we have to add a foreign key. This constraint should consist of local column *"addressID"*, reference table *"AddressBook"* and reference column *"ID"*. This will create connection between *"Orders"* and *"AddressBook"* tables.

At this point we have full database table structure created. Thanks to the foreign keys also their relationships is obvious. This means that we cannot have order items without parent order and we cannot have orders without valid addresses.

Creation of views for reading record rows from a database is a good practice due several reasons. If you are using views for reading a data and stored procedures for data modifications then a potential attacker wouldn't know database structure. The next reason can be if you are using views, you can store more complex queries in database (with large join and group statements) and after that you can use simple ("just only SELECT") queries in application.

So now we should create some views. The first view will be used to retrieve data from *"AddressBook"*. It will have name *"AddressBookVW"* and it will contain query *"**SELECT * FROM AddressBook**"*. The second view will be used to retrieve data from table *"Orders"* and will be called *"OrdersVW*. It will contain query *"**SELECT O.*,A.* FROM Orders O LEFT JOIN AddressBook A ON A.ID = O.addressID**"* for example. The third view which we will create will have name *"OrderItem"*. This view is just only simple view with simple query *"**SELECT * FROM OrderItems**"*.

If we perform all the steps described above then we get ERD with complete database structure definition for our new database project. The database structure should be similar to one shown on in Fig. 9.

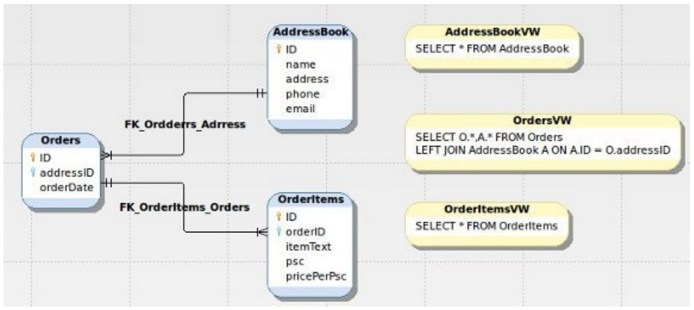


Fig. 9 Tutorial database structure described by ERD

B. *Creation of Database Structure Based on ERD*

If we create ERD diagram describing new database structure, we usually want to write this structure into real database on database server. This could be done by using DatabaseExplorer tool in few steps. At the first we have to create new empty database. The task can be done relevant menu item from context menu shown after right mouse button click onto an opened database connection in the main application panel. The next step is to commit database structure from opened ERD to the new database. This could be done via commit button placed on the toolbar displayed at the top of the ERD editor panel. This button will show commit wizard that helps the user with writing the database structure into selected database. If everything pass well we can see the new database with all new tables and views. In this case the structure should look like one shown in Fig. 10.

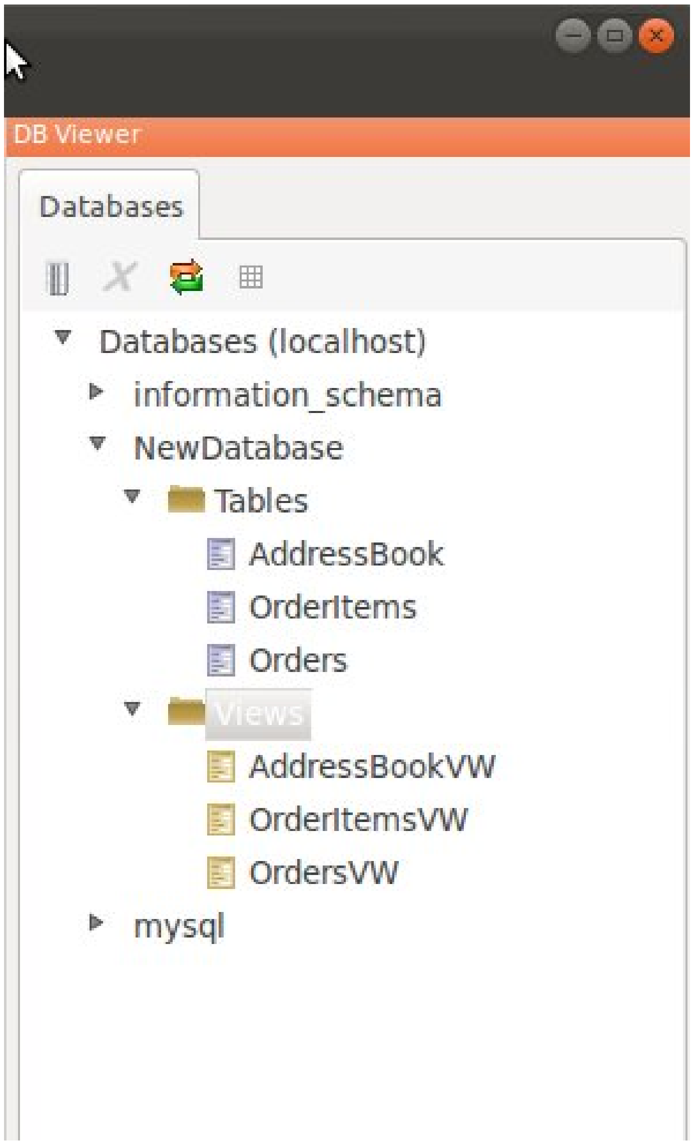


Fig. 10 New database structure

If we want to make some changes in an existing database structure we have a few possibilities. The easiest way is to modify ERD and then perform a new database structure commit. But this could cause a problem. In some cases we haven't ERD defined (if we lost it or if we want to modify an old database).

In this situation we can let DatabaseExplorer to create a new one automatically. This action can be performed by double-clicking onto the database name in the database tree structure, or by selection or relevant menu item in the context menu (shown by right-mouse-button click onto the database name). After that, DatabaseExplorer will create a new ERD and paste all existing objects into it.

Note that one should check the newly created objects because not all of them could be created in a form we expected. More exactly, views could differ from ones previously stored in the database; if we write a view to a real database, the view is evaluated and all non-specific symbols (like *) are replaced by the specific columns names. This would be a problem if we are modifying the table structure which the view points to.

C. *Generation of Database Access Classes*

Until now all the steps were related to the correct database structure definition and creation. Now we will pass to a creation of an application front-end by using CodeLite IDE. DatabaseExplorer integrates with it well and it can produce C++ class prototypes for access the database we can use in our source code. The further work will continue in several steps: at the first we will create access classes prototypes and after that we will use those classes in our wxWidgets application.

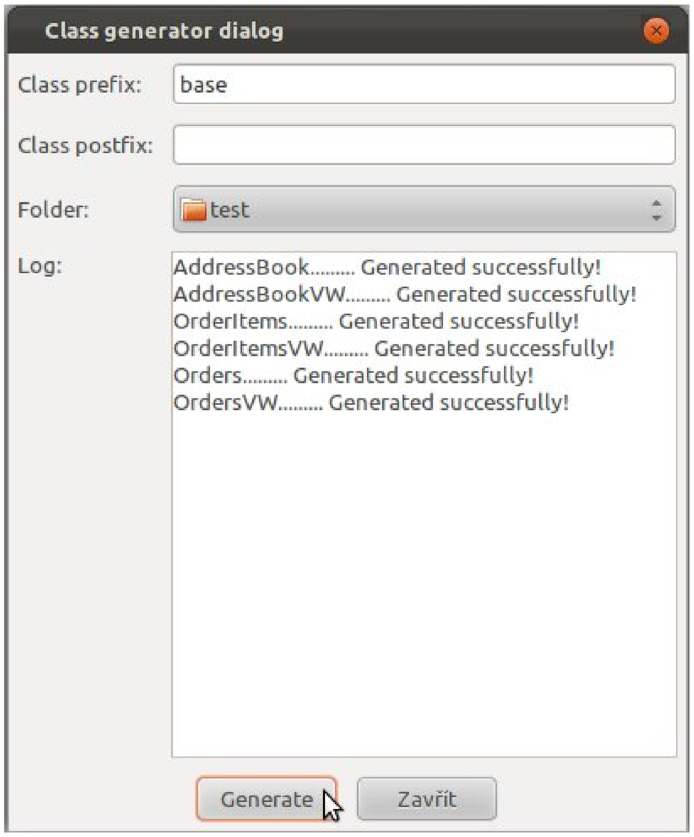
Access class creation is a simple task. In fact we just have to select relevant item in context menu (shown by right-mouse-button click onto the database name). After that a new dialog will be shown. In this dialog we can select *prefix*, *suffix* and a *target path* of generated files. *Target path* is a destination where DatabaseExplorer will create new files. Name of files and name of classes will be compose from *prefix* + table name (or view name) + *postfix*. It means that if we have table named *"Orders"*, *prefix* set to "base" and let the *postfix* be empty then the target files name will be "baseOrders.h", "baseOrders.cpp" and classes names will start with "baseOrders". After clicking the **Generate** button the DBE will generate one file for each view and table in the database structure. Each file will contain few classes. The class with name of database table/view will represent one row from the source table/view. There will be also another class with postfix *"Col"* in generated file too. This class will represent collection of rows described above. We can imagine it like off-line footprint of source database table or view content. If the file is generate for table, there should be also class with postfix *"Utils"*. This class defines static functions for table content manipulation.

Operations done during the code generation are logged in the generator dialog as shown in Fig.11.

Later, generated source code can be easily imported into any C++ application project. In a case we want to modify the generated classes then we should derive new classes from generated ones. In the future, if we will need to modify database structure and to re-generate the classes then our changes made in derived classes won't be lost.

For the testing purposes we will create a simple GUI application containing one main frame window with data grid placed inside it. We can start with simple wxWidgets GUI project template available in CodeLite IDE. After application skeleton is created we have to setup the project to use DatabaseLayer add-on library which is used by both DatabaseExplorer and generated access classes. This tasks is quite common and related to the basic C++ programming approaches so it won't be covered by this paper. Instead, let's focus to database operation tasks.

After successful project setup and the application initialization a database connection must be opened by using relevant DatabaseLayer classes (DatabaseLayer provides a special class for each supported database engine, e.g. MysqlDatabaseLayer for MySQL engine, SqliteDatabaseLayer for SQLite engine, etc.). This should be done in main frame constructor or somewhere else, but it have to be done before we want to load a data from database or change it. Pointer to this database connection object must be passed to generated classes for further processing. At the first, let's insert some data via generated utils classes. As shown in Listing 1, there are static methods named *Add*, *Edit* and *Delete*. This methods can be usually used for direct data modification without need of any SQL query.



```
AddressBook......... Generated successfully!
AddressBookVW......... Generated successfully!
OrderItems......... Generated successfully!
OrderItemsVW......... Generated successfully!
Orders......... Generated successfully!
OrdersVW......... Generated successfully!
```

Fig. 11 Class generator

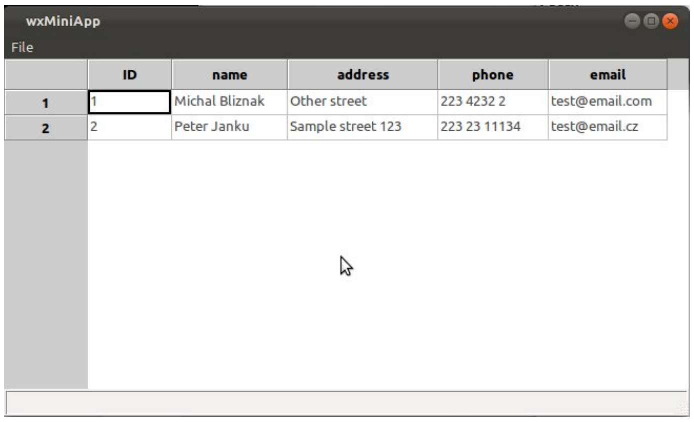
Listing 3 Data row addition

```
DatabaseLayer* pDbLayer = new
      MysqlDatabaseLayer(wxT("serverName"),wxT("
      databaseName"),wxT("user"),wxT("password"))
      ;
baseAddressBookUtils::Add(pDbLayer, 1, wxT("
      Michal␣Bliznak"), wxT("Other␣street"), wxT(
      "223␣4232␣2"), wxT("test@mail.com"));
```

Also other data rows can be added to the database in the similar way. Now let's fill the data grid placed in the application frame with the data. The first step must be retrieving of all data rows collection by using static function *Get* declared in generated class *baseAddressBookCollection*. After that the member function *Fill* declared in the same class can be called for the purpose. Thanks to the *Fill* function the data grid will be filled with the data from the collection. Also, table columns headers are created too.

Listing 4 Filling the data grid with the data

```
DatabaseLayer* pDbLayer = new
    MysqlDatabaseLayer(wxT("serverName"),wxT("
    databaseName"),wxT("user"),wxT("password"))
    ;
baseAddressBookCollection* pOdr =
    baseAddressBookCollection::Get(pDbLayer);
pOdr->Fill(m_grid1);
```



	ID	name	address	phone	email
1	1	Michal Bliznak	Other street	223 4232 2	test@email.com
2	2	Peter Janku	Sample street 123	223 23 11134	test@email.cz

Fig. 12 Sample application containing filled data grid

VII. Conclusion

As shown in the paper the DatabaseExplorer tool is sufficient for the most database-related work and can be used not only for direct database management but as a powerful development tool as well. It is able to visualize or modify existing databases, to create new databases and to generate common production-ready source code for common basic database operations. Thanks to the used distribution license the tool can be used freely for both commercial or open-source/freeware applications development.

VIII. Acknowledgement

The project was supported by the Ministry of Education, Youth and Sports of the Czech Republic under the Research Plan No. MSM 7088352102 and by the European Regional Development Fund under the project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089.

References

[1] DBMS (Database management system) definition. http://www.techterms.com/definition/dbms, 2011.

[2] ERD tutorial. http://folkworm.ceri.memphis.edu/ew/SCHEMA_DOC/comparison/erd.htm, 2011.

[3] ISO - ISO standards - JTC 1/SC 32 - data management and interchange. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_tc_browse.htm?commid=45342&published=on, July 2011.

[4] MySQL :: MySQL 5.6 reference manual :: 13.12 overview of MySQL storage engine architecture. http://dev.mysql.com/doc/refman/5.6/en/pluggable-storage-overview.html, 2011.

[5] MySQL :: MySQL 5.6 reference manual :: 1.3.3 the main features of MySQL. http://dev.mysql.com/doc/refman/5.6/en/features.html, 2011.

[6] PostgreSQL: documentation: Manuals: PostgreSQL 8.1: GiST indexes. http://www.postgresql.org/docs/8.1/static/gist.html, 2011.

[7] Michal Bližňák, Tomáš Dulík, and Vladimír Vašek. A persistent Cross-Platform class objects container for c++ and wxWidgets. *WSEAS TRANSACTIONS on COMPUTERS Volume 8, 2009*, 8(1), January 2009.

[8] Michal Bližňák, Tomáš Dulík, and Vladimír Vašek. wxShapeFramework: an easy way for diagrams manipulation in c++ applications. *WSEAS TRANSACTIONS on COMPUTERS Volume 9, 2010*, 9(1), January 2010.

[9] Joseph Blough. DatabaseLayer homepage. http://wxcode.sourceforge.net/components/databaselayer/, 2011.

[10] Eran Ifrah and David Gilbert. CodeLite IDE Main/Home page. http://www.codelite.org/, 2011.

[11] Peter Janků. DatabaseExplorer homepage. http://sourceforge.net/projects/wxdbexplorer/, 2011.

[12] Robert Roebling, Vadim Zeitlin, Stefan Csomor, Julian Smart, Vaclav Slavik, and Robin Dunn. wxWidgets homepage. http://www.wxwidgets.org/, 2011.

[13] Gerd Wagner. The Agent-Object-Relationship metamodel: towards a unified view of state and behavior. *Information Systems*, 28(5):475–504, 2003.