Overview of the Zeos Database Objects Architecture

Sergey Seroukhov

Merlin Moncure

26 November 2003 (Updated 29/5/2008 by Mark Daems)

Table of Contents

Direct access to SQL databases continues to be a vital technology even in today's enterprise environment. Thousands of two-tier client server applications are developed and maintained in the international business community. Most of them are built off of specially designed application programming interfaces (APIs) to retrieve relational data and execute SQL statements.

Currently, there is several standardized and widely used APIs to access SQL databases, such as ODBC, JDBC, and ADO. Borland also released their proprietary database middleware interface for its development tools, called the Borland Database Engine (BDE). Despite being freely distributed with Borland's popular line of application development tools, the BDE was unpopular because of complexities in installation and poor performance. As Delphi became one of the leading application development tools for the Windows platform, individuals and companies proposed alternative interfaces to the BDE. These “BDE Alternatives” optimized access to the database by directly using the native database driver, providing performance and feature advantages with respect to the BDE.

Realizing the limitations of the BDE, Borland proposed a new type of database interface called dbExpress. This interface was designed to broker access between Delphi and virtually any relational database through 3rd party drivers. Borland significantly improved the performance of dbExpress with respect to the BDE, but the implementation was buggy and supported only a limited subset of SQL that hampered functionality.

The Zeos Database Object component library (ZeosLib) is one of the best-known BDE alternatives. Originally the library was developed for MySQL and PostgreSQL databases, but support for other vendors was soon added. During the development process, certain limitations of the original design became more and more apparent. These limitations began to put a strain on the overall architecture and the development team decided a ground up rewrite was the best way to proceed. The new design was built to handle an extended feature list with several new requirements:

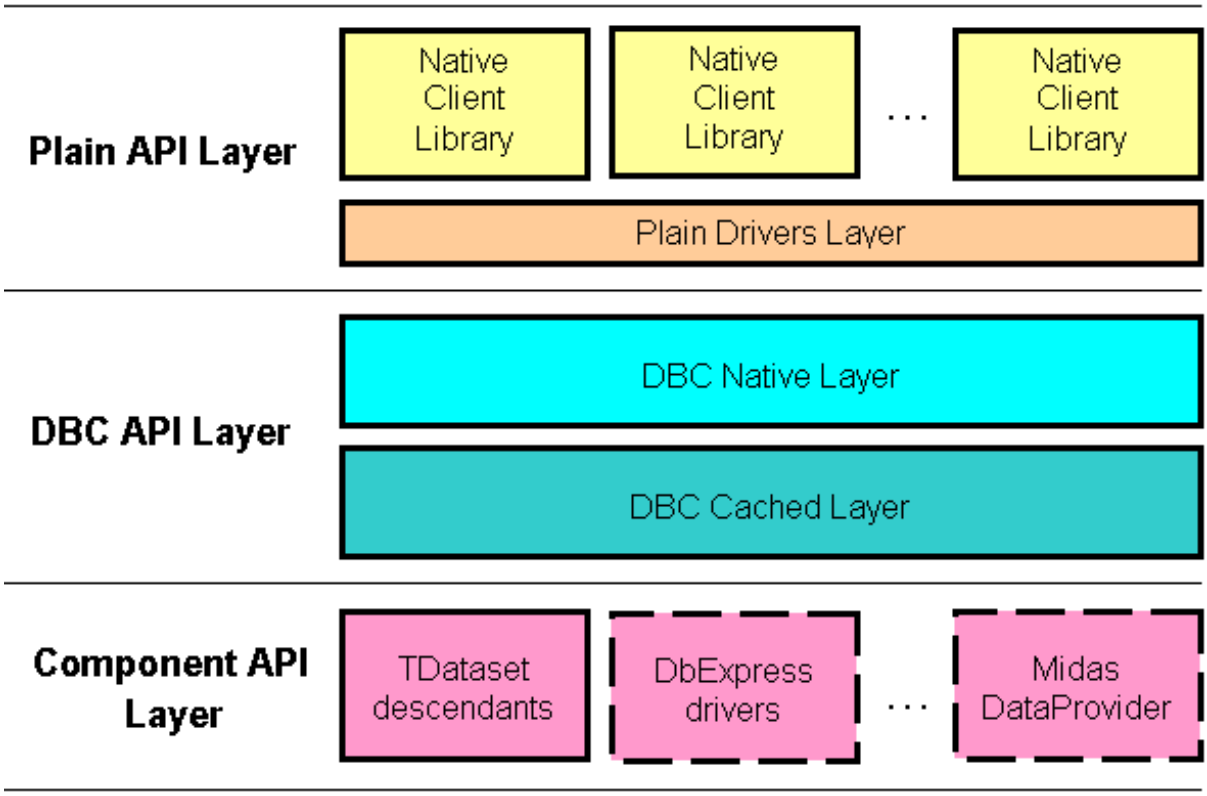
1. Support for different compilers
2. Versioning database driver system
3. “Database Insensitive” design for cross-database development
4. Support for multiple high level interfaces (TDataset, dbExpress, Midas)
5. Extensible feature system for server specific support

1. General Overview

To address multiple and sometimes inconsistent requirements the development team had to completely rethink the new architecture. It is complex with respect to previous designs but not over-designed. Each module put into the new design was carefully considered and carries special unique features. In this article we'll try to present you the new architecture and explain its purpose.

From a top-down view the library is separated into three logical layers:

1. Plain API Layer
2. Database Connectivity Layer
3. Component Layer



The Plain API implements low-level functions specific to each SQL server. The Component API encapsulates the main library interface that is utilized by application developers. The DBC API is the middleware that retrieves, stores and modifies data for the high level components.

Each layer has several horizontal and/or vertical sub-layers semi-independently from each other (see picture 1). Let's go through each layer and look how all that works.

2. Plain API Layer

ZeosDBO components do not communicate directly with SQL servers. Instead they use native client libraries provided with SQL databases. The Plain API layer provides an access to functions of native client libraries, constants and data structures (usually written in plain C) from the Object Pascal language or C++.

That functionality was the original basis for ZeosDBO. Support for multiple versions of client libraries and SQL servers was the main deficiency of earlier designs...

Native library calls (dynamic libraries in Windows and shared libraries in Unix) are represented in programming language as regular functions. For example:

```
ZPlainMySql323.pas:

function mysql_init(Handle: PMYSQL): PMYSQL; external 'libmysql.dll';
```

Usually database APIs do not significantly change between versions. But because function calls do not allow polymorphism adding support for new versions of the Plain API is coding intensive. Hard coded approaches are inflexible and error-prone, which limit long term feasibility.

```
if Version = 'mysql-3.23' then
  ZPlainMySql323.mysql_init(...)
else ZPlainMySql40.mysql_init(...);
```

To implement polymorphism, simplify source code, and provide insulation from changes in SQL server protocols a new extremely thin interface layer was added into ZeosDBO. That layer is called “Plain Drivers” and implemented as follows:

```
// Generic MySQL driver interface
IZMySQLDriver = interface ...
  function mysql_init(...)
end;

// MySQL driver for version 3.23
TZMySQL323Driver = class (TInterfacedObject, IZMySQLDriver)
  function mysql_init(...)
end;

// MySQL driver for version 4.0
TZMySQL40Driver = class (TInterfacedObject, IZMySQLDriver)
  function mysql_init(...)
end;

function TZMySQL323Driver.mysql_init(...)
begin
  Result := ZPlainMySql323.mysql_init(...);
end;

function TZMySQL40Driver.mysql_init(...)
begin
  Result := ZPlainMySql40.mysql_init(...);
end;
```

Using such thin class wrapper allows easy addition of new client interfaces. The specific functionality that requires overriding is encapsulated in the Plain driver. The rest of the code now has a uniform method of providing native database calls, without requiring specific knowledge about the database server.

```
// Initialize plain driver
PlainDriver: IZMySqlDriver;

if Version = 'mysql-3.23' then
  PlainDriver := TZMySQL323Driver.Create()
else PlainDriver := TZMySQL40Driver.Create();

// Use plain driver
PlainDriver.mysql_init(...)
```

With this approach it is possible to use the same API for different SQL servers, independently of version.

3. Database Connectivity Layer

With native access provided to the database, Delphi database aware components can now expect database specific functionality in a uniform manner. However, each SQL server has different semantics that must be designed into the components to provide a universal generic interface. The main goal of the Zeos Database Objects

is to provide this generic interface to the application developer.

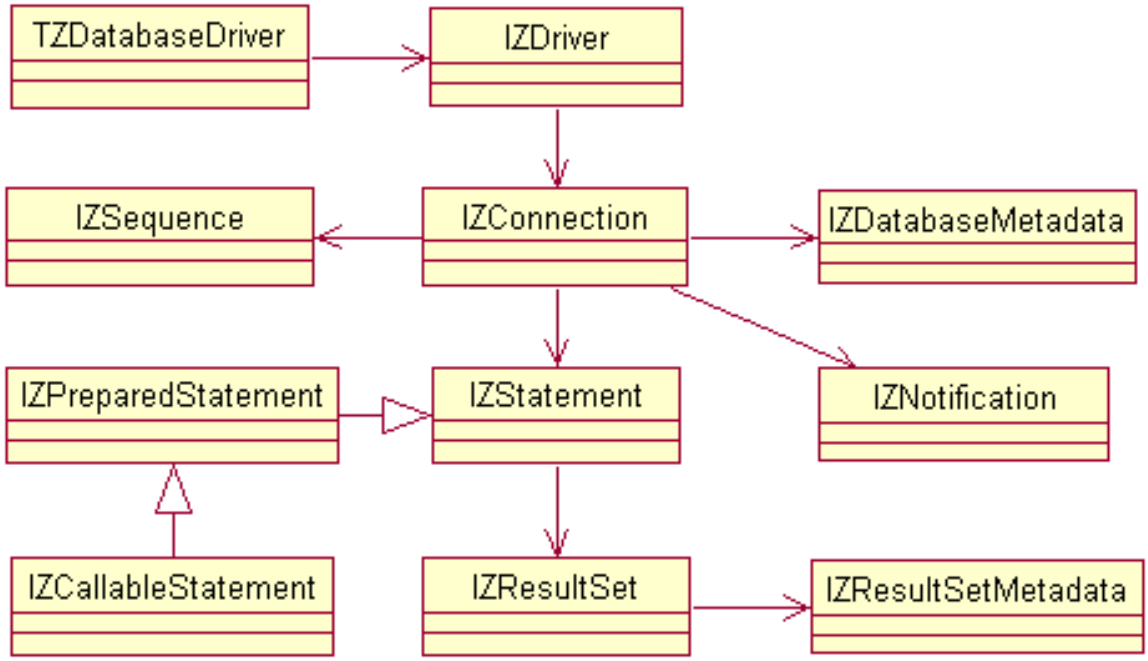
In older versions of ZeosDBO, the intermediate interface was implemented as a class wrapper to the MySQL and PostgreSQL connection objects. These two databases have very similar capabilities so design of intermediate API was not complex. However, support for other SQL servers added new and very specific features. After several extensions to support these features the class architecture became was not clearly defined. Ultimately, the interface broke its encapsulation rules and the high level components were forced to make low level database calls.

To overcome this difficulty in the new version, the overall design was an abstract approach with an intermediate interface. Design of such an interface is not trivial. So to avoid new mistakes it was decided to avoid a proprietary design, and instead draw inspiration from a well-known database API.

As a prototype for the intermediate interface the development team chose JDBC 2.0. JDBC is one of the latest and more popular APIs in that database community. It covers various abstractions such as statements, result sets, stored procedures, blobs, and very rich metadata definitions.

JDBC API is implemented in Java through set of interfaces. Borland compilers support interfaces, so porting JDBC from Java to Object Pascal was straightforward. Data types and method names were generally preserved. Overloaded methods were avoided because of poor support in the C++ Builder family of compilers.

The main DBC interfaces are presented on picture 2.



Standard JDBC interfaces provide a uniform client API. To address functionality specific to different SQL servers, two general approaches were chosen.

1. Developers may initialize database connection object with list of string parameters. Each parameter switches server specific settings. For JDBC that method is not new.

Example: parameters can be defined in Connection URL -

```
zdbc:mysql://localhost/database?compress=true
```

or parameters can be passed to connection factory method

```
Params.Values['compress'] := 'true';
Connection := DriverManager.CreateConnectionWithParams(Url, Params);
```

Additionally in ZeosDBO developers extended other DBC interfaces to initialize other object with specific parameters, particularly Statement object:

```
Params.Values['oidasblod'] := 'true';
Statement := Connection.CreateStatementWithParams(Params);
```

2. In Object Pascal each class is able to implement multiple interfaces simultaneously. We used that to extend standard JDBC interfaces with new methods specific for each particular SQL server. Now each class known not only a standard interface but a server specific interface as well:

```
IZMySQLConnection = interface (IZConnection)
  function Ping(...);
  function Kill(..);
end;

TZMySQLConnection = class (TInterfacedObject, IZConnection, IZMySQLConnection);
...
end;
```

The next step in porting JDBC to Object Pascal was an implementation of cached data access. Actually, many servers provide support for sequential data access only. Caching data on the client side is the important element to implement random data access for the retrieved result sets. On the other hand, many high-level database interfaces use even more sophisticated caching algorithms. So implementation of universal caching algorithms in one place can be a compact and efficient solution.

Cached DBC layer has only few classes:

1. `TZRowAccessor` - organize storage and access to fields of one single cached record in result set (pattern Flyweight).
2. `TZCachedResultSet` - is a cached result set with random data access. It works on the top of another native non-cached result set with sequencial data access (pattern Decorator).
3. `IZCachedResolver` - is an interface to handle special logic to post modified data back to SQL server (pattern Delegator).

The good thing about DBC interfaces that they are so generic, that additionally to regular SQL drivers it's easy to implement special adapters to other database interfaces such as Active Data Objects (ADO) for example.

DBC interfaces are generic, so providing support for additional databases or connectivity layers such as ADO is trivial.

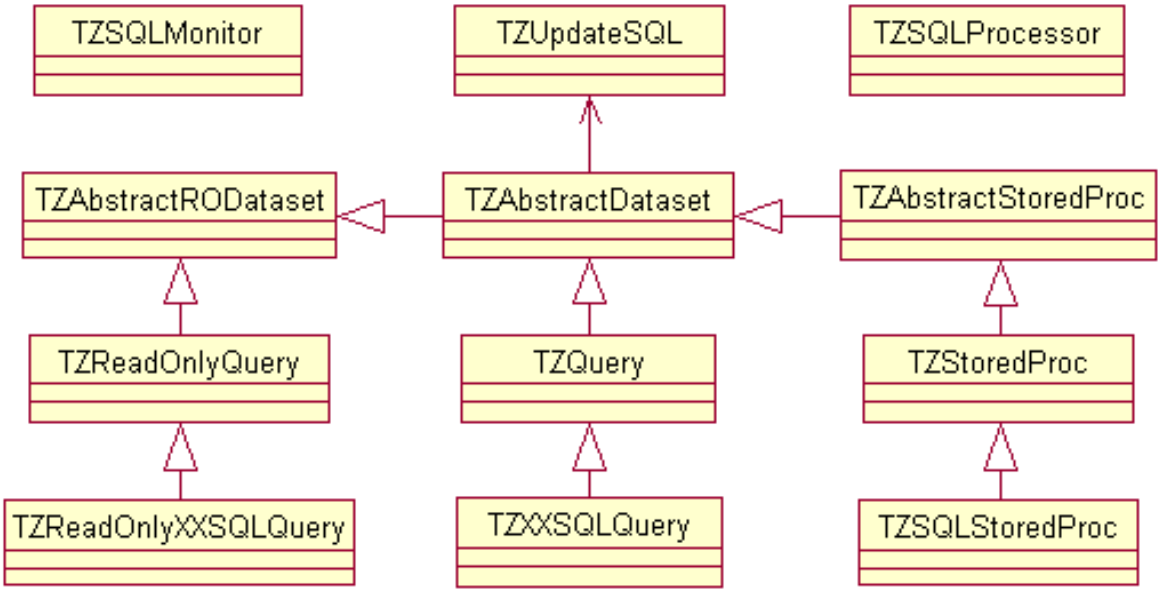
4. Component Layer

The last top layer in ZeosDBO library implements dbware components. These components are used for development in Delphi, C++ Builder or Kylix. Currently Borland compilers support several standards for dbware components:

1. `TDataset` descendent components
2. DbExpress drivers
3. Data providers for multi-tier Midas technology

At the time of this writing this article ZeosDBO library only supported TDataset descendent components. Implementation of other component types is scheduled for future versions.

The component class diagram is presented on picture 3:



1. `TZAbstractRODataset`, `TZAbstractDataset`, `TZAbstractStoredProc` - abstract classes for `TDataset` descendant components
2. `TZReadOnlyQuery`, `TZQuery`, `TZStoredProc` - universal `TDataset` descendant components
3. `TZReadOnlyXXSQLQuery`, `TZXXSQLQuery`, `TZXXSQLStoredProc` - `TDatasets` specific for each supported SQL server. The way how they are propagating server specific functions is described in the previous chapter about DBC Layer.
4. `TZSQLMonitor`, `TZUpdateSQL`, `TZSQLProcessor` - generic auxiliary components

`TDataset` uses maximum the DBC layer functionality to read, modify and store data. Additionally it implements extra functions to filter, search and sort data, connect to visual components and many other things.

5. Epilog

In this article we described the main ideas of the new architecture of Zeos Database Objects component library for native database access. This architecture was introduced in version 6.0 and demonstrated high flexibility and effectiveness. In version 6.1 the code was seriously revised and optimized, but the main principles of the architecture was not changed.

Functionality and flexibility incorporated in the architecture will meet adapt to new requirements in the future. Support for new SQL servers will be added as well as access to SQL specific functionality and support for other high level interfaces like dbExpress and Midas.

To familiarize with the library you may visit the project website at http://zeos.firmos.at or the development page on SourceForge athttp://www.sourceforge.net/projects/zeoslib