Development of cross-platform problem-oriented systems using specifications of database applications *

Alexei Hmelnov[1,2][0000−0002−0125−1130] and
Evgeny Fereferov[1,2][0000−0002−7316−444X]

[1] Matrosov Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Sciences, 134 Lermontov st. Irkutsk, Russia hmelnov@icc.ru
http://idstu.irk.ru
[2] Institute of Mathematics, Economics and Informatics, Irkutsk State University, Gagarin Blvd. 20, Irkutsk, Russia

Abstract. We consider the approach to development of AIS (automated information system) using declarative specifications of database applications (SDA). The specifications of database applications contain all the information about database structure, which is required to build a typical AIS. The information is represented in its pure form, so the specifications are rather concise. The AIS'es are implemented using general algorithms, which are directed by the specifications. We have developed algorithms for such tasks as: user interface generation, query building, report generation, GIS interaction. Using the specifications of database applications and the algorithms the software system GeoARM was implemented. The technology considered was well-tried by use of the system GeoARM for development of several dozens of true-life AIS for different purposes.
In this article we'll describe the approach, that we use for creation of several versions of the GeoARM engine, which use different data access libraries, from the common source code base. The resulting versions of the GeoARM engine allow us to create problem-oriented AIS'es for all the supported platforms from the single SDA.

Keywords: Specifications of database applications · automated information systems · rapid application development · data access technology · source code structuring.

1 Introduction

Automated information systems are designed to accomplish specific information-handling operations [1]. Considerable part of AIS use relational database management systems (DBMS) for storing and processing the information they collect.

* The work was carried out with financial support of Russian Foundation for Basic Research grant #: 18-07-00758-a. Some results were obtained using the facilities of the Centre of collective usage "Integrated information network of Irkutsk scientific educational complex"

Copyright © 2019 for this paper by its authors. Use permitted under Creative Com-mons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2463/paper7.pdf

Usually the database interaction is the central functionality of AIS. In our work we consider database client application (or, shorter, *database applications*) – the AIS'es that implement DBMS user interface. Database client application should allow their users to perform CRUD (create, read, update, delete), search and some other operations, for example, report generation.

The development of database applications in object-oriented imperative languages using class libraries (like, VCL [2], MFC [3], FCL [4]) becomes very repetitive, but still tedious and time-consuming task. Indeed, the code parts, which implement the typical operations for different tables usually have no substantial differences, besides from the names of the used tables and fields. So, there exist approaches, that allow programmers to partially automate the task.

The object-relational mapping (ORM) approach is intended to simplify the code for database interaction. Such ORM libraries as Hibernate/NHibernate [**?**], Entity Framework [6] automates construction of the object model of the database tables. Instead of interaction with tables, the application, which uses the ORM libraries, interacts with the objects. Anyway, it is still required to write the rest of the application code. And, perhaps, when using the data-aware controls, that interact with database without ORM, the rest of the code may be simpler, than with ORM.

Some approaches try to automate the development of the user interface for database interaction (like Model-Based User Interface Development [7]) and the whole application (e.g. Model Driven Architecture [8]). The formal representation of information about AIS structure is used to generate database objects and the code of client application. The generated code is very schematic and requires further development to make it of production quality. As a result it becomes very hard to reflect the changes in the specification, which usually happen during the application life-cycle.

So, the main disadvantage of the modern software development technologies is that they force programmer to write a lot of similar code, which differs only in table and field names. Though certain technologies may help to generate some part of the code, the programmer will have to perform a lot of similar work anyway, and he will have to rewrite all the code to reflect the changes in the structure of the database, which inevitably occur during its life-cycle.

Our approach is based upon the use of specifications of database applications (SDA). The SDA should provide the minimum required information in its pure form about database tables, their fields, the links between them and their usage in the database application. All the other tasks are performed by general algorithms, directed by SDA. We have developed the general SDA-directed algorithms for generation of user interfaces, interactive query building, report generation, GIS interaction, etc., and the program GeoARM, which is based upon the algorithms. The program allows us to obtain a full-featured database application by development of SDA, with the specification being rather small and not containing code duplicates. Some nonstandard tasks can further be solved by plug-in modules, which extend the capabilities of the main application. The approach was considered in more details in [9].

In this article we'll describe the approach we use for structuring the code of our application to be able to create several its versions, which use different data access libraries from the common source code base.

2 Data access libraries

Let us consider the data access libraries, which are of interest for the Delphi/FreePascal developers. We will briefly characterize the capabilities of the technologies from the point of view of the task of the database applications development.

2.1 BDE

The oldest Delphi versions had the only data access library – BDE (Borland Database Engine). Later on its alternatives ADO (ActiveX Data Objects) and dbExpress were introduced.

The BDE development was canceled around 2001. You can still install the library, if required, but it will not understand some important field types introduced since 2001, like that of Unicode strings (NVarChar in MS SQL), bit fields and so on. The BDE has local SQL – the built-in SQL engine, which supports local tables in the files of dBase and Paradox formats. It is now required to change some BDE default settings and the access rights for some folders to support the work with Paradox tables in the modern Windows versions, because in 2001 it was normal for an application to write something to the `C:\` folder or to store its configuration file in the corresponding to the application sub-folder of the `"Program Files"` folder. In spite of all these limitations, sometimes the BDE usage is still the easiest way to implement some database functionality in a small application. That's why we still support the BDE version of our GeoARM engine.

2.2 ADO

The ADO (renamed later to dbGo) library in Delphi is a Pascal wrapper around the similarly-named Microsoft library ADO [10], which is implemented using the Windows-specific technology ActiveX/COM, so the library itself is Windows-specific. The ADO library is still supported by Microsoft and the latest versions of Delphi. It can handle all the known field types and database management systems (DBMS). If some DBMS doesn't have a native ADO driver, it can be accessed through the ADO driver for ODBC via the ODBC driver for the DBMS, which almost always exists. The ADO library is rather effective. The major problem of its usage in Delphi is that the data sets, which use the server-side cursors, have very limited capabilities. So, we usually have to use the client-side cursors, and it strongly limits the size of the tables, that can be handled by the application. The ADO version of GeoARM was implemented second after the BDE version, and it is its most frequently used version by now.

2.3 dbExpress

The dbExpress library is based on the extensive use of unidirectional data sets. This approach may have some advantages and we consider the possibility to implement some day the dbExpress version of GeoARM, but now it doesn't exists.

2.4 FireDAC

The cross-platform development in the latest Delphi versions uses the FireDAC library for database interaction. The FireDAC library works on desktop (Windows and MacOS) and mobile (Android and iOS) platforms. Besides from its cross-platform capabilities the library is very effective. In particular, its data sets load and cache records from database on demand. As a result, it can promptly display in database grid the contents of a table with several million records. The approach considered in this article allowed us to implement quickly the FireDAC version of GeoARM.

2.5 SQLdb

A good alternative to the commercial Delphi IDE is its open-source and free analog – Lazarus. In the Lazarus IDE the SQLdb package is preinstalled. So, SQLdb is the main data access library of Lazarus. We are going to create the GeoARM port, which will use the SQLdb, in the nearest future.

2.6 Other libraries

Besides from the big universal libraries there are many specialized libraries, designed for building clients for a particular DBMS (especially Interbase) through its client API or using the ODBC technology immediately (without BDE or ADO mediation). We consider the possibility to develop GeoARM versions for this kind of libraries.

3 Source code organization

Now let us consider the techniques we have developed to implement several versions of the program from the common code base.

3.1 Conditional compilation

When implementing the ADO version of GeoARM by rewriting the BDE version we used conditional compilation to write the data access technology dependent code parts. Listing 1 demonstrates a code fragment, which uses conditional compilation to get data type names (logical and physical) of a field. However, if we

were continuing to develop the program this way for the other data access libraries, then it would become too hard to understand and support the resulting code.

It would also be difficult to find and rewrite all the places in the code, which should be changes, while implementing the support of a new data access library.

```
{$IFDEF UseADO}
 FTS := GetEnumName(TypeInfo(TFieldType),Ord(FS^.DT));
 Delete(FTS,1,2);
 S := GetADOTypeName(FS^.hDT);
{$ENDIF}
{$IFDEF UseBDE}
 FTS := '';
 S := '';
 if GetPhyTypeInfo(FS^.hType{FldType},FS^.hSubType,FT) then begin
   FTS := FT.szName;
   S := FT.szNativeName;
 end ;
{$ENDIF}
```

Listing 1: An example of data access technology dependent code with conditional compilation.

Therefore we decided to re-factor the code to avoid the conditional compilation and to collect all the data library dependent code in the corresponding modules.

3.2 An abstract data access technology

To get rid of the conditional compilation we have introduced the abstract base class, which describes a data access technology (DAT). Then we have systematized all the conditional code fragments and replaced them by the calls to the corresponding to them methods of the DAT and the other DAT-dependent classes, which we'll describe later. Meanwhile for each of the data access libraries we have created the concrete descendants of the base DAT. When removing a fragment with conditional compilation we move the code from the conditional compilation branches to the bodies of the methods of the corresponding DAT classes. The resulting DAT methods can be grouped into the sections shown in the Table 1.

The DAT class contains the methods, which unify database interaction and database objects handling, which are likely to be used by any database application. Using the classes we can create different kinds of database application: console, GUI, Web-backend.

3.3 DAT-dependent classes

A particular database application contains other fragments, which depend both on DAT and some application task specific modules. Most part of the fragments are used in a particular module only and it wouldn't be effective to place all the

Table 1. DAT class method groups

Section	**Description**	**Examples**
General information	The general information about the DAT, which is required to inform user, which DAT is used by the application, read configuration and so on	ShortName, Description, CfgSecName, Supports
Connection support	The operations with database connections	NewConnection, GetConnAlias, GetConnectedUser, SetDBLoginInfo
DataSet support	The operations with TDataSet (the common ancestor class for tables and views), which can be applied to tables and views	SetDatasetConn, DatasetSetReadOnly, DatasetIsSQLBased
Query support	The operations with database queries	GetBaseQueryClass, NewQuery, QuerySetSQLText, QueryParamByName
Table support	The operations with database tables	GetBaseTableClass, NewTable, TableSetName, TableGetMasterSource, TableSetIndexFieldNames

code into the DAT class methods, because it will make the DAT class module dependent on all the modules of all the applications (GUI-, Web-, console- and so on) simultaneously. Therefore we need some techniques to organize the DAT-specific code in the application-specific modules.

Abstract DAT-dependent classes The most obvious way to organize the code is to make the application-specific classes abstract, and place the DAT-specific code into the virtual methods of their descendants.

We need a way to link this kind of classes to the DAT class to be able to select the right class descendant for the DAT in use. To organize the abstract class descendants we use the utility class `TDATClassRegistry`. We create the registry in each base application-specific module. The modules, which declare its descendants, should call the registry method `RegisterClassFor(<Descendant class>, <DAT class>)` during their initialization. So, to support in an application a particular DAT it is enough to mention the corresponding modules in the `uses` list of some module of the application.

Variators Sometimes it is not desirable to use class hierarchy to represent the DAT dependence. For example, when a class, that depends on DAT, already may have descendants. To be able to vary behavior of the class according to the DAT in use we introduce an auxiliary class hierarchy. Let's call the auxiliary classes

variators. Variator have access to the internal state of the object being extended and may have its own internal state, which holds the DAT-specific fields. The main object should declare a field to hold its variators.

To associate variators with DAT classes we use `TDATVariatorRegistry` – the utility class , which works the same way as `TDATClassRegistry`: variator class should be registered by the call of the `RegisterVariatorFor` method from the module initialization. The method `InitVariators(<Object to extend>, <DAT variator table>)` creates the variators for the `<Object to extend>`, and the method `DoneVariators` allows us to free them.

So, we declare arrays, that can hold several DAT-dependent variator objects. This approach potentially allows us to create an application, which will use several DAT simultaneously. Perhaps, this capability is an overkill, but it worth nothing for us, because we can always use the arrays of one element. On the other hand, if we'll have to support the work with several DAT simultaneously, then it will not require much effort.

On the Listing 2 we can see the changes of the code from the Listing 1 after refactoring. Here the field `FCurDAT` holds the index of the DAT in use (among the registered ones). The conditional compilation was replaced by the call of the corresponding virtual method of variator (the 1st line of the listing). The two alternative implementations are shown in the next two code fragments. Now the code of each implementation is contained in the module corresponding to its DAT.

```
FTS := FVariators[FCurDAT].GetTypeNames(FS,S);
...
function TDBViewFormADOVariator.GetTypeNames(FS: PFldStatInfo;
  var NativeName: String): String;
begin
  Result := GetEnumName(TypeInfo(TFieldType),Ord(FS^.DT));
  Delete(Result,1,2);
  NativeName := GetADOTypeName(FS^.hT.hDT);
end ;
...
function TDBViewFormBDEVariator.GetTypeNames(FS: PFldStatInfo;
  var NativeName: String): String;
var
  FT: FldType;
begin
  Result := '';
  NativeName := '';
  if GetPhyTypeInfo(FS^.hT.hType{FldType},FS^.hT.hSubType,FT) then begin
    Result := FT.szName;
    NativeName := FT.szNativeName;
  end ;
end ;
```

Listing 2: Code fragments from 3 modules of the refactored version of the code from Listing 1

4 Examples of usage

The most observable feature of the FireDAC library is its capability to handle large data sets by loading data on demand. Fig. 1 demonstrates the form for editing records in rather big table. And the table in the Fig. 2 is even bigger. Due to the implementation of the FireDAC version of GeoARM the application is able to work with large tables. The user interfaces shown were generated automatically from SDA. And for the database used here the SDA was also generated automatically using the database meta-information and some heuristics. So, the whole application is a result of several mouse clicks.

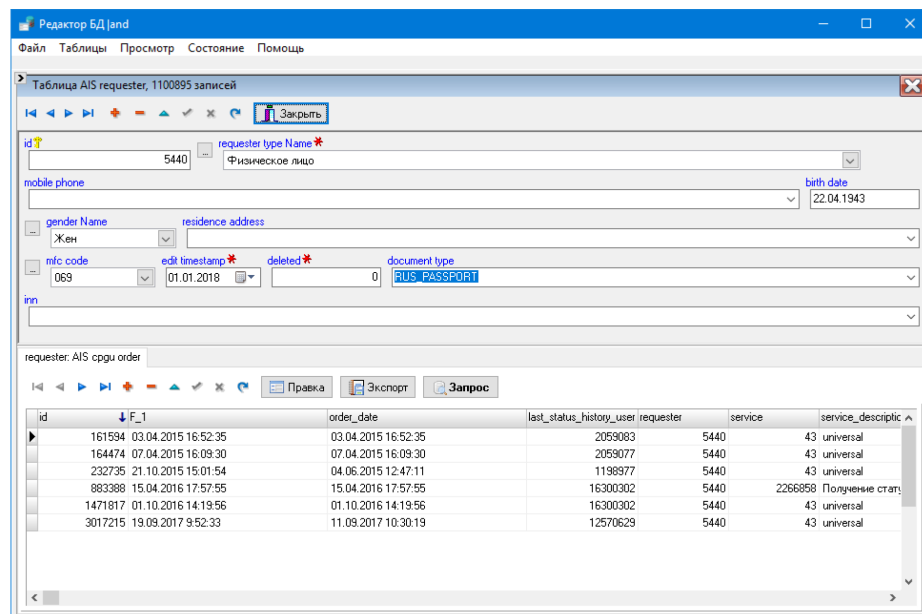

Fig. 1. Automatically generated form for editing records in a big table (it contains more than one million records)

5 Conclusion

We have considered the approach we use for structuring the code of our programs, which use the specifications of database applications (SDA). To get rid of conditional compilation, which would make the code unreadable and hard to support, we use several techniques. The main peculiarities of the data access technology in use are represented by the DAT classes. We also use the DAT-dependent classes and variators to modify the behavior of classes with descendants. Using

Fig. 2. Grid showing a bigger table (it contains more than 49 million records)

the approach we succeeded to develop the version of our program GeoARM for the FireDAC data access library, which can effectively work with large tables.

Using SDA we can quickly develop database applications of production quality.

References

1. ATIS Telecom Glossary - American National Standard, Automated information system (AIS) https://glossary.atis.org/glossary/automated-information-system-ais/. Last accessed 22 Sep 2019
2. VCL Overview http://docwiki.embarcadero.com/RADStudio/Rio/en/VCL. Last accessed 22 Sep 2019
3. MFC Desktop Applications https://docs.microsoft.com/en-us/cpp/mfc/mfc-desktop-applications?view=vs-2019. Last accessed 22 Sep 2019
4. .NET Class Library Overview https://docs.microsoft.com/en-us/dotnet/standard/class-library-overview. Last accessed 22 Sep 2019
5. Hibernate ORM documentation http://hibernate.org/orm/documentation/. Last accessed 22 Sep 2019
6. Entity Framework (EF) Documentation. / Microsoft. Data Developer Center. https://docs.microsoft.com/en-us/ef/ef6/get-started. Last accessed 22 Sep 2019
7. Paulo Pinheiro da Silva, Tony Griffiths and Norman W. Paton. International Working Conference on Advance Visual Interfaces 2000 (AVI2000)// Generating User Interface Code in a Model Based User Interface Development Environment. Palermo, Italy. 2000. pp. 155-160.
8. Frank Truyen. The Fast Guide to Model Driven Architecture http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf. Last accessed 22 Sep 2019

9. Bychkov, I.V., Hmelnov, A.E., Fereferov, E.S., Rugnikov, G.M., Gachenko, A.S.: Methods and Tools for Automation of Development of Information Systems Using Specifications of Database Applications. In: 3rd Russian-Pacific Conference on Computer Technology and Applications (RPC), pp. 1-6. IEEE, Vladivostok (2018). https://doi.org/10.1109/RPC.2018.8482170
10. ADO Programmer's Guide for using ADO Objects https://docs.microsoft.com/en-us/sql/ado/guide/ado-programmer-s-guide