Firebird White Paper

Why typical Delphi projects often fail ...

A subjective observation by Holger Klemt, October 2014

Many of you have known me for some years and some of you originally got to know me as a Delphi developer, who regularly presented this platform at lectures at various conferences.

I started with an early Delphi 0.9 beta version in 1995 and was immediately impressed. At that time the Internet was not yet omnipresent, Windows 95, Compuserve and AOL however were setting things in motion. At that time, trade fairs such as the CeBIT in Hannover and the Systems in Munich were still the principal events, to equip yourself with important information.

In search of viable alternatives for the database applications that I had already created for customers based on Turbo Pascal, dBase and later Microsoft Access, Delphi offered me from the outset everything I needed for my projects. I had mastered Turbo Pascal some time ago, so Delphi was naturally my first choice.

Some interesting new opportunities arose in collaboration with the then very active Borland Distributor better office. In the years since 1996 I was also part of the Borland CeBIT team and had presented each new Delphi version in the software cinema to unbelievable crowds. Further lectures at various conferences around the world were great fun and the positive response of the audiences was enormous.

It really only required a few minutes to convince any audience of the performance and comparatively unbelievable productivity. Create a new project, add TDatabase, connect to DBDemos; add TTable, connect to a table; add TDatasource, add TDBNavigator, include TDBGrid, generate the TField objects using the TTable field editor and drag these using drag and drop into the form, compile, start, wow! Audiences often stood open-mouthed and wide-eyed in front of us and could hardly wait to buy the newest version of Delphi, no matter the cost.

The sales figures at that time were certainly impressive, I however had nothing to do with that side of things. I was, so to say, a show programmer and not a salesman. From my perspective at the time, this development environment was the non plus ultra and I enjoyed demonstrating and presenting the existing techniques in various trainings and lectures worldwide. During this time I made many contacts, of which a large number still exist today. Through consulting and programming assignments I have been able to actively participate in a range of customer projects. The majority of these projects are still in use today.

At this time however software companies, development departments and individual developers had already entered a blind alley with this component- and form-based system, which could have been avoided using flexible architecture.

That was the golden age of the component producers and many a software house at the time used all components that were currently considered flavors of the month. Components were used and integrated, etc.

New Delphi versions also always required new versions of the components, so that many updates were only possible if the component manufacturer also provided an update.

As long as the component manufacturers earned sufficient income from the Delphi environment it was not a problem. Many component manufacturers were also active in the Beta Test newsgroups, so that the components were generally available by the release date or shortly thereafter.

After also finding myself in this dead end in the first two years, I shifted over towards implementing my projects using a general architecture, which I have often presented at numerous developer conferences under the title OOP.

As a programming software architect I then implemented this approach in various long-standing customer projects. This core technology is still in use today, and in part, continues to be independently and actively developed by our customers.

The above-mentioned example, which we presented at the CeBIT and Systems trade fairs actually already illustrated a large part of the problem. We used it at the time for demonstration purposes, because it resulted incredibly quickly in a usable prototype.

Dependencies

The database structure is defined at database level using a suitable database editor, i.e. the customer table is defined, for example, with a table field for the zip code, behind which a zip code table is specified as a foreign key. The zip code is often a compulsory field. Some have become a little more sensitive since the German reunification, as four-digit postal codes were changed to five digits along with the ability to allow leading zeros. So instead of an integer data type a text field had to be defined. In Delphi a form or frame is then created for exactly this structure, with TDataset, TDatasource, TDBEdit and TDBLookup ComboBox on it, and various events are hung behind the datasets and finally, the field types are connected directly in the field editor.

An advantage of this approach is that a result is obtained incredibly quickly, namely a usable application for the end user, with real forms and a real database connection. Unfortunately, there are not only customer addresses, but also supplier addresses, employee addresses, etc. As a result, the appropriate table must be recreated with the same tools for each form.

Events with the same tasks are spread throughout the source code in various units. Techniques such as form inheritance promise simplifications, but these can unfortunately result in having the opposite effect. The outsourcing of TDatasets on data modules disburden the items on the form, however result in many cases, especially in a team, new problems.

Such projects tend to fail to meet simple requirements when it comes to globalization. Who can imagine that there are for example no zip codes in Ireland, so that a compulsory field in this case needs to be replaced by a dummy. The whole process, constructed and based on previous knowledge, must now be adapted in all the forms. The database structure also needs to be adapted to meet the new requirements. Whoever amends the zip code fields from **string[5]** to, for example, **string[10]** and defines them as a "non-compulsory field" at the same time, meets according to Wikipedia the global requirements for zip codes. They will however quickly discover that any entries in the TDataset field editor will continue to expect the old length and cannot be opened without an error message. Similarly, the column widths in TDBGrids need to be adjusted in all the forms to meet the new requirements.

The programming team is busy for days making adjustments and testing the new requirements. Software vendors also need to prepare a suitable method for a rollout without any data loss.

The problem with many Delphi applications constructed in this way was is the direct static dependency of the user interface on the database structures. Even trivial alterations, such as the increase of a field size on the data structures is not taken over by the Delphi program. These are distributed in diverse references across many forms and have to be individually adjusted at each location by the developer.

Software architecture part 1

In my EKON OOP sessions around the turn of the millennium, held in hopelessly overcrowded rooms, some visitors even having to sit on the floor at the hotel in Moerfelden as I am sure some of you will remember, I demonstrated a simple mechanism by which large parts of this problem could be avoided.

The central focal point here was a small Delphi program, which generated large parts of my source code based on the table structures. For each table there was a separate, automatically created unit, which mapped all table fields in a class. A constantly expanded base class, from which all automatic classes were derived, ensured the implementation of reads, writes, etc. This was the sole location for references of all the database components used. This could not only be easily replaced, but worked even using only simple tools on more exotic platforms such as the IBM AS/400 iSeries, without developers having to worry about any specifics.

Almost all SQLs were generated automatically. Furthermore, the code generator then generated for each table an additional derived Extended Class in a separate unit, which was only created once. The developer could then override methods in these units, since these were not overwritten by the code generator. The code generator was restarted after each database alteration and the Delphi compiler then quickly disclosed anything that was no longer compatible.

Standard tasks, such as reading records, writing, deleting, viewing or selecting, were implemented by the base class. References to individual TxxGrids or similar were only made here, so that subsequent alterations were possible at any time. In the Extended Class complex forms overwrote all relevant methods and replaced them if necessary with standard forms. Therefore even classically created forms also could take on partial tasks, but could also access the derived classes when implemented.

This sounds more complicated than it is. The source code generator has grown over the years with the base class and was usually extended by customers themselves. The advantages of this method soon became apparent for a customer, where the source code generator created approximately 1600 automatically generated error-free interface units for over 800 tables at the simple click of a button.

The solution to the above problem could therefore be settled very quickly using this architecture, the tables were changed in the database, the code generator restarted and the interface units immediately updated and all globally-defined rules matched the data model. The newly compiled source code handled the basics immediately following the database modification.

The disadvantage of this solution was the initial relatively long development time, which was required for the base class and the source code generator, in order to implement all requirements. Once the system was created, the development time relativized.

The advantages, to later be able to dynamically respond to any database modification, led to part of the development team continuing to develop the database model solely in cooperation with the specialist departments, without having to program even a single line of Delphi themselves.

The source code generator and the Delphi command-line compiler were quickly started using a batch file, so the department could then, at the click of a button, pass the new or adapted form, integrated into the overall system, over for testing. This often led to the department making further requests for alterations, which were also integrated into the database model and led to the next prototype, or resulted in new requests being made to the development team, which could be implemented in the Extended Class.

Practical use

This code generator technology developed and refined at the time in a long-standing customer project is, to my knowledge, still in use today at this and other customers, and still continuously extended. Porting the complete software to a new version of Delphi was quick because the majority of the components and methods were rapidly dealt with by making adjustments in the code generator and in the base class.

Of course this method limits the flexibility within the respective forms to a certain extent, but it prevents the greatest curse of all Delphi programs: everything is somehow interdependent. If the database does not fit the form, there are problems; if the forms do not fit each other, there are problems etc.

Fundamental changes can only be realized in typical Delphi projects with a great deal of time and effort, and often the new functionality causes new problems in established modules.

Daunting examples

If software companies' customers nowadays request customization, packages are often resorted to as a solution. Personally I don't hold much with such solutions. This is certainly my subjective opinion, but even with my years of experience I still arrive at the same result today. The record holder that I have personally met during my consulting work had a CRM software with about 2,000 installations, with 80 different customer versions created by compiler switches. The complete system was totally unmaintainable and customers constantly complained about the faulty software and about recurring errors in newer versions, which had already been fixed, and missing customizations which had been overlooked in the new versions.

The company owner, by his own admission not a programmer but a very good salesman, relied on the judgment of his principal architect, who convinced him for years that there was no better solution.

During a code review day at the customer site, we quickly determined that this was not a general problem with the tool used, but rather a self-made problem.

As in this case, I have in the course of my consulting work, particularly in the Delphi environment, met very many, often highly-qualified professionals with excellent knowledge of the processes of their respective industries, who have taught themselves programming by reading numerous books. Some very good industry solutions have emerged from this. The simplicity of the Pascal language allows it to be quickly understood and learnt. Many a time I have heard "I learned a few programming basics back in school/at university, so I then just went with it".

In this way a number of successful companies have emerged who, with their excellent process mapping and a suitable and functional business software, are the ideal partner for many customers. Many of them are however facing major challenges 10 or 15 years later in order to remain successful in their markets.

Customizing

Unfortunately the basic application often becomes more and more complex, due to individual large customers, who approach the software manufacturer requesting the implementation of special requirements and wishes, so that many easy-to-use business solutions increasingly develop into a complex functional colossus. The result is that customers then tend to continue to use the outdated version of the software.

The usual approach in at least 80% of all Delphi software houses is to produce a complex application with central Delphi source code and a central database structure. This must always be the same for all customers, because there isn't a functional variant mechanism. The remaining 20% mostly use techniques such as packages to meet individual customer requirements at least in certain parts. However, the dependencies here are only slightly less complex than with compiler switches or other approaches.

Reengineering and version changes

Unfortunately both customers and processes change, so that many software companies fail to meet requirements at some stage. Unicode ability was introduced in Delphi 2009, but some of the older components were unfortunately incompatible. At this time, the component manufacturers' market had already declined significantly, so that many components were no longer updated and therefore could not be used in the new Delphi environment.

For this reason, entire projects have ended up totally stuck. These projects cannot be converted to new Delphi versions in the foreseeable future, because many components are linked across the whole code. Replacement of an old, well-established component library causes significant restrictions for many software houses and may even demand a completely different GUI.

In such a case the question arises for any software company whether it is really worth investing several man years in the development of a new version based on a new Delphi version, just to perhaps upset many customers in the end by having to introduce a new GUI.

Even a company like Microsoft has lost many Office and Windows 8 friends, particularly long-standing users, who are now confined to older versions or have switched completely to alternatives.

From my perspective here is where the criticism of and the failure to understand Borland and Embarcadero begins. The new Delphi versions try to make even the most simple component libraries incompatible, even if these actually are 100% compatible with newer versions. Artificial obstacles are constantly being put in the way, which prevent using any existing project with all its components in a new Delphi version. Whether it is due to compiler version numbers, changed units, changed call interfaces or types. Even for simple projects using foreign components, the cost of the conversion is not to be underestimated. As Delphi now appears as a new version almost every half a year, the task of compiling a project in each latest version is just simply enormous.

Software architecture part 2

Four years ago we began a new ERP project for a highly innovatively-thinking customer in the metal and plastics processing industry, as our customer could not warm to any of the existing solutions on the market. There were several reasons for this. First and foremost, the majority of the solutions were too complicated to use, too expensive and in many areas too inflexible.

Since our new project, BRP-Software, was to be used as a basis for other customer projects as well as for our own internal order processing and license management, many points were clear from the outset:

- The client should have functionalities similar to that of a web browser, based on a suitable set of rules, to display the database contents individually.
- The business logic is all in the Firebird database and has no place in the client.
- The data model is based on the use of specific internally-developed system tables from which the client can flexibly obtain the information necessary to create the data masks.
- The same client can be used for different databases and depicts the input forms on the screen depending on the database content.
- The client should be cross-platform natively deployable without having to resort to any tricks. Platforms are 32- and 64-bit Windows, Linux and Mac OS-X.
- Native support for the Firebird database was essential for all platforms.
- Development and testing must be achieved virtually simultaneously.
- There is no list of specifications; instead all business processes are designed and tested in the system as required and, where appropriate, incorporated in the real system on the same day.
- Components will only be used if they are already available in the IDE on all platforms.
- Graphical functions are preferentially implemented by OwnerDraw routines.

The current version of Delphi 2010 and the new XE version at the time were still missing several features, so following numerous tests we decided upon Lazarus. Both Delphi versions were still pure Win32 platforms, although Win64 machines were already widely used and an x64 version of Delphi had been announced a long time ago. We had already gained initial positive experience using Lazarus in smaller client projects, especially with 64-bit applications.

For our new product, BRP-Software, we were able to consolidate our wide experience of recent years. Especially important for us was the choice of IDE and also the selection of any potentially viable components. We did not want to make any of the mistakes that we had encountered in customer projects in the past.

At the same time, from my point of view, an integral component of the new software was the ability to work like a web browser and to be served by a Firebird server at the opposite end, which would provide the data model as well as the business logic and procedures. All visual components are generated dynamically.

There are therefore completely different tables and processes in the IBExpert customer database as in our customers' databases. While licensing is an integral element for ourselves, our customers have nothing to do with licensing at all. This table therefore does not exist in their database. The article master at IBExpert also differs significantly from the article master, which we have implemented for an individual project for a client in the logistics industry.

As soon as the user starts the BRP-Software, his rights are verified in the connected database, and all corresponding modules enabled for this user are displayed on the screen. When the user switches to article administration, the corresponding mask is displayed for the database table, depending on table structure and metadata.

We or the customers themselves can make individual adjustments to the database. The BRP browser immediately incorporates such adaptations.

In the BRP Enterprise version for example, an extremely flexible time-keeping model was implemented in collaboration with a customer. The acceptance among employees is very high. The data accurately records the real working times accordingly, so that production planning and costing has access to continuously up-to-date and accurate data on the production status.

For other customers this is perhaps much too complex or even too simple. We or the customers themselves can make individual adjustments to the system.

Multiplatform

The key factor in our choice of the Lazarus IDE is the fact that with a very clear architecture we have selected a suitable programming language with which we can implement this architecture onto the required platforms. We certainly could have created a similar solution with .NET-based languages. In our case however, our experience gathered over 20 years of working with Delphi/Lazarus/VCL/Pascal would only be of limited help. Moreover .NET is only multiplatform in Windows.

The number of security vulnerabilities that have meanwhile been found in Java convince me that I have chosen a good platform.

In the current Delphi version multiplatform is only available with FireMonkey for the supported platforms, which I reject for several reasons. The critical question that Embarcadero needs to ask themselves is why their own IDE cannot be implemented on anything other than the Win32 API. There appears to be something wrong with their concept.

Lazarus as an alternative

For those who have developed with native Lazarus on Linux or Mac OSX or even on the Raspberry Pi, will know why multi-platform GUI development only makes sense with native IDEs on the respective platforms. My appreciation of the hype surrounding cell phone applications is limited to the view that it is purely a way to make money. We rely on web applications that can also be used on Windows Mobile and other platforms. Using jQuery Mobile, about 30 lines of PHP and an Apache/Firebird server, it is possible to program extremely effective database applications for mobile use, which are up to date and run stably even with several thousand users. Unfortunately, even Apple tends to be rather user-unfriendly with its consumer applications.

Due to compatibility problems, we still use Delphi 5 for IBExpert development today, because porting the 1.7 million lines of source code to a newer Delphi version would, in our point of view, be tantamount to a complete new development, without achieving any significant advantages. We are still struggling today with the global architecture tailored to the Delphi version and components used at the time when we started IBExpert back in 1999. The project originated as a classical Delphi project with many forms and with units and with many complex concatenations.

Should we at some point reach the absolute limits of Delphi 5, we will certainly develop a successor based on Lazarus in order to continue using large parts of the existing source code. Yet another advantage is that we also have access to the entire Lazarus IDE source code, so that we can fix any annoying bugs in the IDE itself if necessary.

Why don't all Delphi projects fail?

Good or bad results can be achieved in any programming language and most IDEs. Although "good" does not necessarily mean "wonderful". From a business perspective, the productivity factor is far more important. If I get paid 10 man days for a project by the customer, and the project can be implemented in 5 man days, it is definitely good; if I need 20 man days, it is not so good. If I cannot realistically estimate the expenditure when calculating the price, but actually only guess how much time I might need for the project and then prepare my offer accordingly, this is an approach that is rarely successful in the long term.

The advantage, and at the same time the largest disadvantage of a Delphi development is, that even without big architecture a software can be created which can still satisfy the user. This is however often realized very ineffectively from a programming perspective. It is also possible to lose yourself completely in the architecture, without delivering any usable results. Source code with extensive use of generics and other specifications is not necessarily better than one without. And poor source code is rarely improved by the use of generics or other techniques.

This is no different in Lazarus. The biggest advantage with Lazarus is the durable compatibility to older versions and to other platforms. Those who do not want to repeatedly reinstall the IDE, because bugs are only fixed in new, fee-based versions, will appreciate Lazarus. Also for the reason that Lazarus can even be operated from a removable hard drive or a USB stick at any time, because the paths are not stored somewhere in the registry. Search paths to new components are detected automatically via the LPK packages.

In my opinion I estimate a maximum of 10% of software vendors who rely on Delphi, have a flexible and powerful architecture in use, with which all the developers are satisfied. The remaining 90% are more or less stuck in a blind alley, because fundamental dependencies prevent real innovation. Many customers only see a migration to Visual Studio as a possible way out. But that is by no means the only option, as this means a completely new development. Without a suitable architecture this will end up back in the same blind alley in 5 to 10 years.

For developers, the introduction of a genuine and flexible architecture, both on the basis of the Delphi version already used, and on the basis of Lazarus or other platforms is, in the long term, much more promising than the switch to a different platform, for which primarily the complete know-how needs to be established. We are happy to help.

BRP technology in practice

Simply contact us at info@ibexpert.com or call +49 4408 359349 so that we can arrange an appointment to show you, free of charge and without obligation, in a remote support session lasting 60 to 120 minutes, just how our system works and how you can program the system yourself.

We offer different licensing models for our technology, suitable for the use of the complete manufacturing and ERP software in house, and also for system vendors who wish to use it as a basis for the establishment of their own specific business solutions. Basic Firebird SQL skills are all that is needed to make customized adjustments.