FFF97 – Oberon in the Real World

Dr. Josef Templ
Software Templ OEG

Abstract

The Oberon programming language and system, subsequently called the Oberon technology, are well known in the software research community. Few applications, however, exist outside academic institutions. This paper describes one of them (FFF97), which constitutes a database front end application that has been in mission critical use by more than 25 concurrent users since 1998. FFF97 makes full use of Oberon's innovations including dynamic loading, commands, garbage collection, type extension, integrated text and form system, component technologies, model/view separation, low resource consumption, etc. It is claimed that choosing the Oberon technology and a particular approach to software construction, known under the term 'Prototyping', was of vital importance for this project, which, as one of the rare examples of custom software development projects, was completed within the scheduled time and budget without compromising quality or functionality.

1. Introduction

The successful application of research results to real world problems can be considered the ultimate justification for spending money on pure research and as a proof of the correctness and relevance of the research results. In the field of software engineering, one controversial research project was the Oberon project [1], carried out by N. Wirth and J. Gutknecht at ETH Zurich starting in the late 80s. Many computer professionals still doubt the advantages and relevance of this technology because it started from scratch and ignored industry standards, and because it appears to be too simple to compare favorably against industry standard technologies. This paper describes the story of FFF97, a database front end application based on the Oberon technology, that has been in mission-critical use by more than 25 persons since 1998. FFF97 embodies a perfect example of applying the results of the Oberon project to the real world, not because it shows that the problem could equally well be solved in Oberon, but because it shows that the problem could *better* be solved in Oberon. In fact, we are not aware of any other system that would have been suited better to this particular task.

In the following chapters we will describe the application domain, the chosen approach to software development, the rationale for choosing the Oberon technology, architectural and implementational aspects of the developed system. We will also present a summary of vital points and some general remarks.

2. The Application Domain

FFF97 is a tool to manage funding for applied research projects by an Austrian governmental organisation (FFF, Forschungsförderungsfonds der gewerblichen Wirtschaft). Activities include registration of all sorts of received postal matter (most notably applications for funding), grading applications with respect to technical, economic and other properties, requesting additional or missing information, maintaining groups of applicants for joint applications, association of FFF staff with individual applications and subtasks, maintenance of funding proposals and funding decisions, maintenance of a calendar of monthly presidential decision sessions, communication with applicants, answering ad-hoc queries for applicants, automatic or semi-automatic production of printed letters to applicants, automatic reminders for various purposes, full text search in text documents, printer management, checking access rights based on user roles, keeping track of travels to applicants and other travels, generating address lists for a large volume of letters, maintaining variables for current interest rates for various purposes, maintenance of third party funds and communication funding proposals and decisions with third parties, grading the project outcome and monitoring how money is spent, carrying out payments, maintaining liabilities, automatic statistics for the yearly funds report and for accounting purposes as well as for the distribution of workload among the FFF staff, automatic account creation and booking in an electronic accounting system, automatic payments in both directions, etc.

The above list is by no means complete and is continually extended with new services and refinements to existing services. However, it should be sufficient to show that it is a real world problem and it is non-trivial by the sheer amount of different activities. At first glance it appears to be a typical database application where one of the many existing 4GL tools would provide the perfect development environment. However, complications arise when the target platform and environment is taken into account and the need for text integration is investigated further.

3. The Target Platform

FFF97 was supposed to run in an existing Windows PC network, featuring Intel i486 CPUs at 33 or 40 MHz with 16 MB of main memory. The operating system being used was Windows 3.1 with a Novell file server. In 1996 this was no longer on the cutting edge, but quite common in offices that cannot afford to upgrade all their PCs every two years. Upgrading the CPUs was not possible for technical reasons, because the low-cost motherboards were simply not compatible with any CPU upgrade available then. Given the tough hardware limitations, 4GL tools were ruled out completely, because they tend to create bulky and slow solutions as can be seen by the project history.

4. The Project History

FFF had been using a Philips P4000 database system with dumb terminals for applications management and accounting parallel to a Windows PC network for other typical office activities. It decided to move to a unified PC solution in order to

- upgrade to state of the art client-server technology,
- get rid of the dependency on no longer supported P4000 systems,
- increase the speed of database operations,
- eliminate the duality of two completely incompatible systems (P4000 and the Windows PC network) on every desktop,
- provide for more flexibility in respect to new services,
- improve client interaction, and to
- improve integration of text documents.

A first attempt by a subsidiary of an Austrian bank corporation to create a solution based on Oracle Forms4 failed due to unbearable response times and other user acceptance problems. The system used Microsoft Word as a text component and was able to create texts with database information included, which was quite reasonable in principle. For unknown reasons, however, it took several minutes and thousands of database requests until a simple data input form opened on a user's desktop PC. This behavior, among other problems, could not be repaired, so the solution was abandoned and the project was started again from scratch.

5. Restarting From Scratch

Given the tough hardware limitations and previous experience with Oracle's 4GL tool reported to us, we decided to rule out 4GL tools altogether and focus on an efficient and innovative general purpose programming system available then under the brand Oberon/F from the ETH spin-off company Oberon microsystems [2]. Oberon/F (later called BlackBox Component Builder) incorporated all the innovative ideas of the Oberon project in a form that was well suited for Windows PC users. In particular it used standard Windows look and feel (overlapping windows, menus and other GUI elements), supported Windows file systems and provided a database interface. Dynamic loading, command activation, garbage collection, an efficient compiler for a slightly extended version of the Oberon language and, most notably, text integration were preserved. Given these prerequisites, we estimated the initial development work as two man-years and the development time as one calendar year.

6. Prototyping

In order to avoid repeating the mistakes of the failed predecessor project it was decided to base the new project on prototyping. This means that there is no written detailed specification of the functions to be implemented. Instead, the developer creates a prototype of a functional unit based on a vague specification of the tasks to be performed. The working prototype is presented to the users and leads to more detailed requirements based on user feed-back. This takes into account that it is virtually impossible to specify the requirements of a complex system in detail and has the advantage that the user sees what (s)he will get long before the project is finished, thereby avoiding unpleasant surprises. An unbiased judge is installed that both parties (developers and users) agree to call if there is no agreement in the termination of the feed-back loop. The decision for one particular offer among several companies was also made based on prototyping a subset of the required functionality in order to demonstrate the availability of adequate development tools and skills. For more details on prototyping see [3].

A few years later, refined variants of this approach to software development became known under the term *Extreme Programming* [8], which is based on four guiding principles in order to embrace change: Communication, Simplicity, Feedback, and Courage. Since we used the term prototyping throughout the project we shall stay with this term in the following.

7. Integrated Development and Execution Environment

A rapid application developement system (RAD) is the technical foundation of any prototyping approach. Obviously, slow compilers and linkers as well as bulky executables are contradictory to prototyping. Even state of the art integrated development environments (IDEs) such as Visual Studio or Delphi are not sufficient. They provide a graphical user interface and sophisticated wizards for various common tasks, but under the hood they are little more than point-and-click interfaces for traditional command line tools and do not exceed the traditional application development model. They still need to link an application even if linking can make use of dynamic link libraries (DLLs). In order to run a modified version of a program, the program has to be rebuilt and started from the main entry point. Any internal program state between multiple executions is lost. Oberon’s idea of an integrated development *and* execution environment (IDEE) makes it much more efficient to perform incremental changes to a program. Only the modules that have been changed are reloaded, the rest of the program is retained and preserves its state. Furthermore, any kind of repeating activity can be automated by means of user-defined commands, which can be executed from within the IDEE.

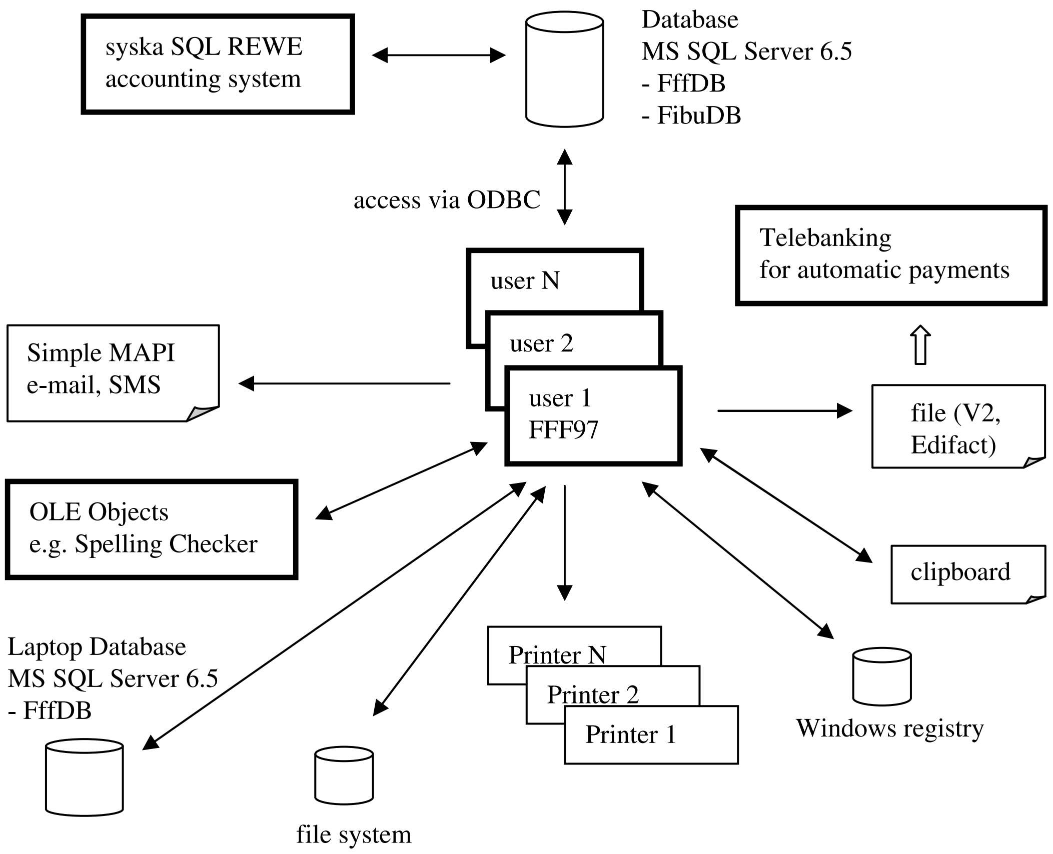
With BlackBox’s form subsystem it is even possible to make many user interface changes by simply editing the form (just like editing graphics). There is no code generator involved. The form is stored as a document in a file and can be reloaded and used from there. This idea is based on the Gadgets[4] user interface toolkit of the ETH Oberon System 3 project and turned out to be an invaluable simplification of the development process. It clearly separates user interface concerns from the functionality of a program and permits real time GUI changes interactively with users, without even restarting the application.

Another aspect of rapid application development is the production of documentation in the form of online help texts. We used BlackBox's integrated text subsystem for writing help texts (as well as program texts) which avoids the need for external tools that make it slow, inconvenient and unreliable to create online documentation, especially for a large project.

In summary, with BlackBox we essentially end up with an "IDDEE", an integrated development, documentation and execution environment, which goes beyond what standard tools support.

8. Component Interaction

The following illustration gives an overview of FFF97's interaction with other components.



There is a common database server for both the FFF projects database and the database of the accounting system. This simplification has become possible by selecting an accounting system that is based on a client-server architecture (syska SQL REWE [5]) and supports Microsoft SQL Server [6], which we also managed to connect with FFF97. It allows us to access both databases from within FFF97 using only one database connection and even allows SQL commands that access both databases within one command. A second database connection is used for optional selective export of data to a local laptop database for users who use a laptop when travelling to applicants. So far, modified data cannot be imported from a laptop database, since this would require additional version control mechanisms (merge replication) not provided by MS SQL Server 6.5.

A standard telebanking program is used for carrying out automatic payments. The interface is simply a file that follows a particular syntax (called V2 or „Normdatenträger“ and in the future will be an EDIFACT interchange consisting of PAYMUL and DIRDEB messages).

FFF97 can access any Windows file system and load and store files in various formats (ASCII formats, RTF, etc.). It can also import and export data using the Windows clipboard and can print to any Windows printer.

The Windows registry is used to store default settings per user, such as configurable window positions, settings for automatic logins, printer configurations, etc. The Windows registry turned out to be a rather volatile database, especially when Windows is run in a Novell network, because with nearly every Novell upgrade or reconfiguration (carried out by a professional network administration company) all registry settings have been lost so far.

Since the FFF staff is distributed across multiple offices on multiple floors, each user has access to different printers and FFF97 allows each user to configure printers for various kinds of paper, e.g. blank paper on Printer 1, letter paper on Printer 2 and so on. Whenever a document is printed that is known to need letter paper, the preconfigured letter printer (defined by printer name, input tray and orientation) is provided as a default, which speeds up printing and avoids using the wrong kind of paper. Printer selection is not part of the standard BlackBox libraries, so we had to go through the intricacies of the Windows API for this purpose.

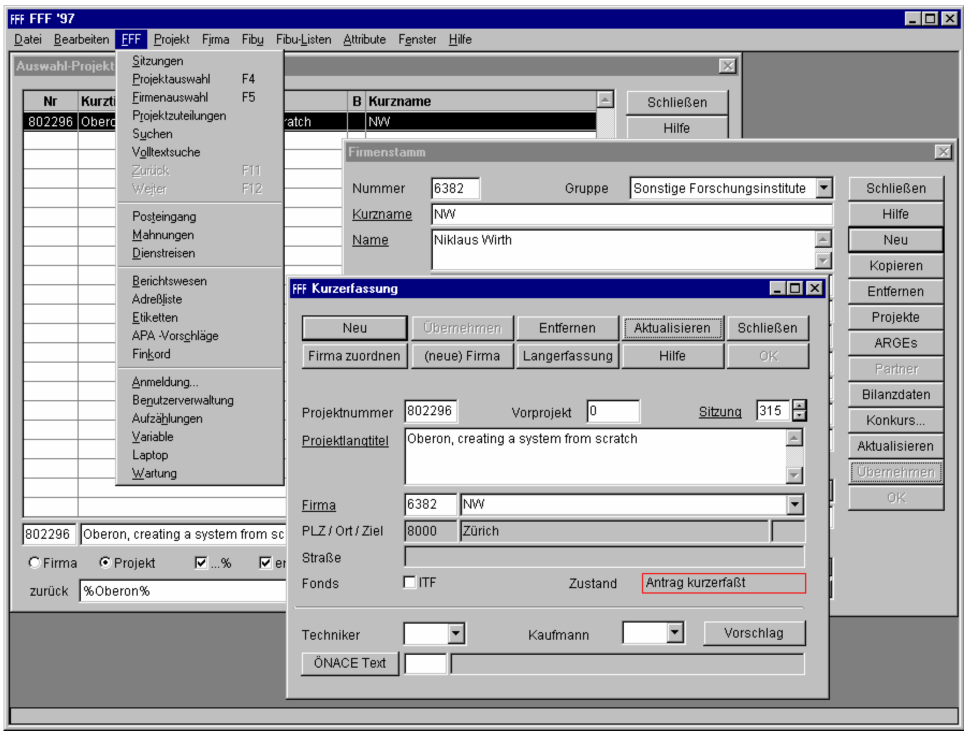
BlackBox supports integration of COM/OLE objects and we made use of this for connecting the Microsoft office spelling checker. This worked in principle but requires one of the Office Applications (e.g. Excel) to be loaded in order to get access to the spelling checker. The result is some waste of memory, startup delays, memory leaks (inside Excel) and slow communication with the spelling checker. The built-in BlackBox text subsystem also supports integration of OLE objects, which we used mainly for bitmap objects, but which the users also use sometimes to include Excel spreadsheets or other objects.

9. FFF97 Program Architecture

FFF97 is a 32 bit Windows application which runs under Win 3.1 (in the Win32s subsystem) and under Windows 95/98/NT. Under Win3.1 it uses 16-bit ODBC drivers that run under a ‘generic thunk’, under Windows 95/98/NT it uses 32 bit ODBC drivers. The syska SQL REWE accounting system also supports ODBC, but in 1996 it was a 16 bit application which needed 16 bit ODBC drivers also under Windows 95/98/NT. This caused conflicts with 32 bit drivers in various situations. Figuring out working combinations of server OS, client OS, client applications and ODBC drivers was a hard task at the beginning and took weeks of trial and error.

At the level of the development system there is a subsystem called fff, which is represented by its own root directory at the same level as the text, forms, sql or development tools subsystem. A running application is the combination of all available subsystems. The directory fff contains subdirectories for the source code, the object code, symbol files, resource files, documentation and more. For the task of migrating legacy data, we introduced a second subsystem called P4000.

Any user of FFF97 has zero or more roles that allow access to selected parts of the program. The roles are maintained within FFF97 by a system administrator, which again is a special user role. Currently there are 28 different roles that can be freely combined. FFF97 logs in to the database server using a generic login name and password. In addition the user types his FFF97 login name and password that is used to determine his role(s) in FFF97 and control his access rights. We deliberately do not rely on the much less flexible database features for access protection, thereby avoiding dependencies on the database server product.



Nr	Kurzti	B	Kurzname
802296	Oberc…atch		NW

802296 Oberon, creating a system from sc

○ Firma ● Projekt ☑ ...% ☑ er

zurück %Oberon%

Nummer 6382 Gruppe Sonstige Forschungsinstitute

Kurzname NW

Name Niklaus Wirth

Projektnummer 802296 Vorprojekt 0 Sitzung 315

Projektlangtitel Oberon, creating a system from scratch

Firma 6382 NW

PLZ / Ort / Ziel 8000 Zürich

Straße

Fonds ☐ ITF Zustand Antrag kurzerfaßt

Techniker Kaufmann

From a user‘s point of view, FFF97 appears as a Windows MDI (multiple document interface) application, which allows the user to open any number of (sub)windows in an application window. Of course, it is also possible to start FFF97 more than once, in which case multiple application windows are opened at the expense of starting multiple application processes. Subwindows typically are of one of two kinds: (1) input forms, and (2) text documents. Both kinds allow non-modal use of the system, i.e. it is possible to switch from one window to another or to activate menu items at any time. There are exceptions to this rule, however, which appear in the form of requests to the user to answer a yes/no question or to acknowledge a message. Such dialog message boxes are, in good old Windows tradition, modal.

An unexpected consequence of making input forms non-modal was the disabling of keyboard accelerators for command buttons. Normally, under Windows, you can define a keyboard shortcut for a button, which allows you to activate the button without leaving the keyboard. This is not possible with non-modal windows due to Microsoft’s user interface conventions.

10. Data Model and Legacy Data Transfer

First of all, we had to deepen our knowledge of SQL, which obviously would become of vital importance for the implementation of the system. We also had to familiarize ourselves with the peculiarities of MS SQL Server 6.5. This includes server installation and administration and SQL language extensions and limitations. We learned most of the more subtle points on demand as we proceeded with the design and implementation of the new relational data model. We deliberately did not look at the legacy data model on the P4000 at this time, but designed from scratch and migrated the old data model to the new one using scripts with SQL-statements at a later stage.

Exporting the legacy data from P4000 turned out to be a rather slow process, because the only way was to use a serial 19200 baud link and it took two days to complete data transfer. We had to implement data format conversion routines in order to be able to import the exported legacy data into SQL Server tables. After that, we had access to the old data model by means of a 1:1 representation of the old tables in the SQL Server and could apply scripts to convert the old data to the new data model. By doing that, it turned out that the old data was highly redundant and contradictory as well as incomplete, meaningless or incorrect. We had to prepare scripts to repair the data in many places in order to get useful data for our new data model.

We decided not to do a full transfer of accounting data to the new accounting system but to transfer the accounting plans and balances only, which is quite common when switching from one accounting system to another. We also had to split several accounts to make them compatible with the new data and accounting model. Most of the accounting data transfer was done automatically, i.e. only a few accounts had to be opened or corrected manually. Fortunately, accounting data was much more consistent and less redundant than project data, so we could transfer the accounts with a lot less development effort than the project data.

All in all, the Microsoft SQL Server 6.5 turned out to be a stable product with few surprises and was fairly easy to learn and manage even for a novice. The most unpleasant thing is probably that empty strings are stored as a string containing a single blank character (fixed in 7.0), which implies that all potentially empty strings must be trimmed not only before writing but also after reading. It was possible to normalize the new data model (with very few exceptions) without running into performance problems.

11. SCP - SQL Client Pages

One of the fundamental requirements of FFF97 was the need for text integration in the application. It should be possible to type in texts, store them in the database, generate read-only texts from database contents (reports) and generate writeable texts from database contents (e.g. templates for individual letters with the letterhead and parameterized standard texts filled in automatically). All these requirements are essentially variations on the same theme, coping with text, and this is where Oberon (and BlackBox) really shines. Based on the integrated text subsystem it was easy to provide for something that might be called SQL Client Pages (SCP) in analogy to Java Server Pages (JSP) or Active Server pages (ASP). We therefore defined a simple syntax that allows us to mix a text with SQL-Select statements, which are executed, and to substitute text variables with fields of the result set. The statement syntax is SQL, the SELECT statements are executed on the database server and the text expansions and substitutions take place on the client. The following simple template should illustrate the mechanism:

```
Clearing Code    Name
<@SELECT code, name FROM Bank ORDER BY code;
<code>           <name>
>
```

This defines a report consisting of underlined column titles and an ordered list of pairs of all the banks contained in the table *Bank*. *code* is the clearing code of a bank, *name* is its name. The construct <@SELECT ... > opens a scope for field names (code, name) and a range for text expansion. The character “;” separates the SELECT statement from the text section. The text section is instantiated and appended for every row of the result set and within every text instance the values of the fields of the current row are available as <variableName> (<code> and <name> in the above example). The text may contain arbitrary formatting attributes and embedded elements, which are preserved during instantiation. It should also be noted that the mechanism may be nested recursively, i.e. within a text section there may be another <@SELECT...> phrase and it is allowed to use the variables of the enclosing SELECT within an enclosed SELECT statement and text section. There is some additional syntax for formatting numeric fields, date values and other special formats, for inclusion of files, for conditional text expansion, and, most notably, there is an extension mechanism in the form of an arbitrary command, whose output is embedded in the text.

```
<@COMMAND M.P param1 param2 ... >
```

This construct calls command M.P and replaces itself by the result produced by M.P, which writes text (including arbitrary elements floating within a text) into a text buffer. Examples include inserting bitmaps for scanned signatures, inserting hyperlinks or generation of non-static header or footer elements into the generated text.

We provide several ways of activating SCPs, including opening the resulting text in a read-only text window, in a writeable database text window, or as a view object that is not immediately opened in a window but may be further processed by the caller (e.g. as an e-mail attachment). The files containing the SCP specifications are called *templates* and are stored in a separate directory.

It might be argued that it should also be possible to use one of the many commercial report generators for the task of generating reports. While this is true in principle, it would be a tremendous waste of resources, introduce extra license costs, complicate report definition, limit the report structure to RPG-style reports, preclude an extension mechanism, complicate the inclusion into an application and serve only for read-only reports. The task of creating letter templates would still require a special solution. We have also observed that other programs switch to custom report generators after experimenting with standard tools (the syska SQL REWE is a prominent example for this).

12. Customized Visual Components

During the course of the project it became evident that the predefined visual objects, such as text editor, input controls, etc. were not sufficient for our specific needs and that customized components would become necessary.

Normal BlackBox texts, for example, are stored as a file and contain little more document information than the location and file name. For storing texts into a database there must be additional information available in the form of a database key which allows modified text to be replaced at a later stage. We therefore extended the standard text views by our own wrapper class that introduced the missing instance variables (database key, document type and paper type). This turned out to be a straightforward process, although it necessitated some delving into the details of handling visual components and working with wrappers, which are the preferred method of introducing subclasses in BlackBox. There were more visual components to be created, most notably a data grid component (based on a simple prototype provided by Oberon microsystems) which serves to display a database query result set and permits navigation and selection of entries. Other visual components include an editor for grading projects with respect to certain properties, attributed straight lines, and specialized date and currency input controls.

BlackBox provided all the prerequisites necessary to create such components and it was never necessary to switch to a different development system in order to create such classes.

13. Meta-Programming Facilities

The original Oberon system provides meta-programming using the concept of commands alone, which are exported procedures without parameters. BlackBox component builder goes beyond that and provides full access to variables and commands with parameters comparable to the one in [7]. Two important applications exist: forms with user interface components and the database interface.

The forms subsystem allows the linking of user interface components to program variables and procedures. A text input field, for example, is linked to a character array or an integer variable and displays the contents of this variable. This makes it very much like a 4GL programming system, because it allows a program to access the values contained in user interface components as normal program variables. Optionally, two procedures, a notifier and a guard may be linked as well. Both constitute commands with parameters. The notifier is called whenever a particular event, such as typing a key, occurs and may trigger related actions. The guard is called in order to inquire about the state of a component, which may be disabled, read-only, or normal. The forms subsystem and not the application program is responsible for activating guards and notifiers, which simplifies programming significantly. Without such a mechanism it is hopelessly difficult to realize self guiding forms, i.e. forms that guide the user by disabling user interface components that are currently not applicable.

The BlackBox database interface allows the programmer to deal with whole records, not just with individual fields, for reading query results and composing SQL commands. When reading a row of a result set, a record variable may be used that matches the structure of the row. After reading, the record fields contain the values of the (positionally) corresponding fields in the result set row, which again would otherwise require a 4GL programming system. The Sql-subsystem is not hard coded in the BlackBox runtime system but simply uses the meta-programming facilities provided by BlackBox in order to traverse record variables and to fill in the field values in an appropriate format. When executing an Sql-command, the notation :M.V, where M denotes a module and V denotes a variable, may be used in order to expand a record to a comma separated sequence of record fields. This saves a lot of typing especially in Insert statements, as for example in “INSERT V VALUES (:M.V)”.

In both examples the runtime overhead introduced by using the meta-programming facilities is marginal, the benefits for the application programs are huge.

14. Statistics

The following list contains some key measures about FFF97.

- Source code: 86 modules with a total of 1,3 MB, 21.000 Semicolons
- Templates: 167 files with a total of 758 KB
- Resource files: 102 files with a total of 520 KB
- Documentation: 100 files with a total of 594 KB
- Object code: 86 files with a total of 1.2 MB
- Executable process size: about 6 MB
- Databases FibuDB and FffDB: 160 MB each, about 70% filled
- Number of text documents in FffDB: 19.000
- Number of database objects: 39 tables with a total of 450 fields, 1 View, 1 stored procedure.

15. Summary of Vital Points

Let us now summarize the points which we believe were vital for the success of the project.

- Prototyping turned out to be possible to circumvent the writing of detailed specifications, which nobody is able to accomplish for a complex project. It depends, however, on efficient tools and the close cooperation of all participants. The short feed-back loop avoids developments in the wrong direction, so we never had to throw away big pieces of work. Without an Oberon style IDDEE tool, we believe, it would have been impossible to achieve the required number of development cycles.
- Error avoidance is of vital importance for a prototyping approach, since long debugging times simply cannot be afforded. Oberon's approach of using an airtight type system and automatic garbage collection turned out to be an invaluable asset even in the world of commercial data processing. The procedure activation stack dump provided in case of a runtime exception and the possibility of inspecting global variables were sufficient for debugging logical errors. This is, by the way, more than one gets in traditional IDEs, since Oberon does not need to make a distinction between release (without debugging) and debug version.
- Efficiency of the resulting programs in terms of time and memory consumption can only be achieved with a compiled general purpose programming language and reasonably designed libraries. Both is the case with *BlackBox*, which made it possible to run the application on i486/33Mhz with 16MB main memory. It is still an advantage after switching to bigger PCs, because it allows the user to start more than one FFF97 process or other office tools in order to perform certain parallel tasks. It is also an advantage when working offline on a Laptop, which must host a database server locally. A small disk footprint is also an advantage when starting a program via a network from a central file server.
- Oberon's dynamic loading strategy turned out to be of three-fold benefit to us: (1) it avoids the extra linking step during development, (2) it allows the application to be installed on a central file server and to be loaded incrementally with almost unnoticeable loading delays even on a 10Mbit network with about 30 active users,

and (3) it allows us to maintain and update the system incrementally via remote access over a single 64 Kbit ISDN line.

- Text integration pioneered by the original Oberon project and continued by BlackBox turned out to be at the very heart of our application. A simple SCP report generator needed just a few weeks of coding and allowed us to create text templates for a broad range of applications including reports and letters with little effort and great flexibility.
- Meta-programming support in BlackBox introduces many features of 4GL programming systems and allows a tight integration of the program with databases and user interface components. There is no extra coding required and there is also no code generator involved.

16. Concluding Remarks

The following contains some thoughts about lessons learned during the course of the FFF97 project. Some of the points are actually statements made by Prof. N.Wirth that proved to be more than true in practice.

- “Keep it as simple as possible, but not simpler” (Einstein, often quoted by Wirth) is especially important in a prototyping approach in order to be flexible enough to incorporate additional or deviating needs. Whenever it is unclear what the user really needs, do not invest a big effort in a preliminary solution.
- A user interface should be created using visual design tools, not by means of lengthy statement sequences. There is no need for a code generator within such a form editor, which simplifies development and the development system significantly. It also provides a clear separation of concerns, GUI design on the one hand and functionality behind the buttons on the other. Programmed user interfaces are only justified in exceptional cases, where dynamic layouts are needed.
- The lack of static typing in the database world constitutes a major problem for client applications whenever the data model needs to be changed. It is impossible to test all dependent programs systematically. With the common approach of using dynamic SQL inside a general purpose programming language, this situation cannot be expected to change in the near future.
- Modern database systems and programming interfaces exhibit a degree of complexity that is hard to digest for someone used to the simplicity of the Oberon system. Most of the complexity stems from irregularities and exceptional cases that would better be omitted from the specifications. This might free some resources for introducing higher level concepts such as types and type extension into a database management system, which is still a long way off.
- Wasting resources is never a good idea even if the resources appear to be unlimited, as is the case with disk space nowadays. With a highly normalized

data model and compact representation of text documents we were able to fit the database into less than 160 MB, a small fraction of a modern hard disk. There are advantages, though. Backup is fast and our database fits almost entirely into the main memory of a modern server computer, where 256 MByte and more are the norm. The same argument holds for every level of the memory hierarchy.

- Compiler construction is at the very heart of computer science. Even in a commercial data processing application, one can make good use of a simple language and a recursive descent parser as can be seen by the SCP example.
- Custom software projects such as FFF97 are always at the technological limit. With improved hardware and software technology user expectations are raised, because standard software created with thousands of man-years raises the standard of comparison to new heights. Keeping up to those expectations is everything but easy considering the limited development resources for custom software.
- What a user wants is not always what a user needs. Taken literally, the wishes formulated by users range from trivial to unsolvable. They often include *how* to do it, based on a particular idea, and not only *what* to do. It is the developer's task to find the intersection of realizable and powerful concepts, which in many cases deviates significantly from what the user originally wanted.
- Sometimes the user is not able to distinguish between "always" and "often" and between "never" and "seldom". This can easily lead to oversimplifications that do not take exceptional cases into account.
- Oberon microsystems Inc. did an excellent job of hiding the complexities of the Windows API and preserving the spirit of Oberon in a Windows environment. There are some points left to be improved (e.g. advanced GUI interface elements), of course, which are mainly due to limited manpower. The complexities and irregularities of the Windows API make it difficult to integrate Windows services and components seamlessly with a reasonable effort.
- The FFF97 project would certainly not have been possible in 1996 and would still not be possible in 2000 with the much hyped Java technology. Although Java resembles Oberon in some aspects (e.g. garbage collection, strong typing), its library is far from being useful for this kind of application. It would require at least five times the memory, execution time, development effort and program startup time, let alone the unsolved problem of text integration and printing.

17. Acknowledgements

Without the contribution and support of many people FFF97 would not have been finished successfully. I wish to thank my business partner and brother Dipl.-Ing. Erwin Templ, with whom I had the pleasure to work on this project. He was always a highly motivating and inspiring discussion partner both in solving problems of detail and for developing the big picture. Prof. Gustav Pomberger supervised and coordinated the project and played the role of the unbiased judge. Thanks to his wise project management, we rarely had reason to call him. The FFF management deserves respect for taking the risk of charging a young, small and unknown company with the development of their mission critical software. The FFF staff was a patient and constructive prototype tester and did its best to communicate its needs to us. Susanne Litschauer shouldered the burden of organizing meetings and writing reports and was always a competent and friendly contact person. For the rare cases where we needed help or bug fixes Oberon microsystems Inc. responded quickly. I also want to thank Hanspeter Mössenböck and Eva Jaksch for proof reading this paper. Last but not least I want to thank Prof. N. Wirth for the technological ground work he laid together with J. Gutknecht in the Oberon project and for the privilege of working with him at ETH Zurich.

References

1. N. Wirth, J. Gutknecht, Project Oberon – The design of an Operating system and Compiler, Addison Wesley, 1992
2. Oberon microsystems Inc, http://www.oberon.ch
3. G. Pomberger, L.J. Heinrich: Prototyping-orientierte Evaluierung von Software-Angeboten., Theorie und Praxis der Wirtschaftsinformatik, Heft 197, Sept. 1997
4. A. Fischer, H. Marais, The Oberon Companion, VDF Hochschulverlag AG, Zürich 1998
5. syska SQL REWE, http://www.syska.de
6. Microsoft SQL Server, http://www.microsoft.com/sql
7. J. Templ, Meta-programming in Oberon, PhD thesis, ETH Zurich, 1994
8. Kent Beck, Extreme Programming Explained: embrace change, Addison Wesley 1999