Structured Gui Language (SGL) Programming

Frankfurter Xbase++ Tracks 2015

Roger Donnay

Xbase++
the business language

POWERED BY
alaska
software

Contents

How Do I Get To “Hello World”?

In this session, Roger Donnay and Bobby Drakos are hosting a joint tutorial on an alternate methodology for developing large and small applications much faster using linear programming techniques. Bobby will explain the issues and problems he encountered trying to understand the new paradigm of object oriented programming, event processing, UI (User Interface) design, screen management, and Xbase parts when attempting to move his application from a text-based, SDI (Single Document Interface) system written in Clipper, to a state-of-the-art MDI (Multi-Document Interface), multi-threaded, GUI application that would rival and eventually out-class his competition in the New York Taxi management industry.

Roger will explain the reasoning behind the development of this SGL (Structured Gui Language) methodology stemming from his encounters with the failed RAD (Rapid Application Development) of the 1990’s. Attendees of this session will receive the entire eXpress++ 2.0 package including source code. This is a free license. There are no strings attached.

- In this session the following concepts will be covered:
- What is Declarative vs Imperative programming architecture?
- What tools are needed when working with Xbase++ for the first time.
- Advantages of the pre-processor and how it improves programming productivity, code readability and software maintenance.
- When, why and how to apply code abstraction principles to the design of GUI windows.
- How to write code that is self-documenting.
- What is data-driven programming and why has it failed?
- Where does your intellectual property reside in your application?
- Why today's application development requires a community.

A little History

No self-respecting programmer would consider it a great achievement to write a Hello World program in any language. This would require an admission that he/she is a “beginner”, yet we have all been beginners from time to time over our programming careers. Before being introduced to the world of digital computing I spent 8 years in the U.S. Navy as an electronics technician specializing in radar and communication systems. In the Navy I worked on "analog computers". They were very sophisticated devices that could calculate range, altitude and direction of targets, yet they needed to be constantly calibrated. I was always frustrated with the inaccuracies of such devices and so when I was introduced to the world of digital computation, I was immediately hooked.

Programming for me started in 1969 at Burroughs Corporation. I knew nothing about assemblers, compilers or interpreters in those days. I had no access to disk drives, tape drives, ram, rom or the like. The term “software” was not even used in those days and the microprocessor hadn’t even been invented yet. My task was to write a set of instructions that would cause a piece of hardware to write the clock-track on a fixed-disk drive system. The set of instructions was written in a series of 1’s and 0’s and given to a technician whose job it was to solder a diode in a matrix PC board for each 1 and leave the diode out for each 0. That was my first “program”. From there I went on to write programs, still using 1’s and 0’s, onto cards which would be used to burn ROMs. This was my introduction to the first Intel microprocessor, the MCS-4. It was a 4-bit processor with only 16 instructions. This program controlled the reading and writing of a data on a cassette tape in a hand-held terminal used for collecting inventory data. My first introduction to using an “assembler” was writing computer terminal software using the Z-80, an 8-bit microprocessor. From there I went on to write an MRP system in BASIC, followed by another in dBASE-II for my own computer accessories business. I found that I was better at programming than at running a manufacturing business so I chose to follow that path and so I evolved to dBASE-III, then Clipper and finally to Xbase++ as my primary language.

So what is my point here, you may ask? Why the biography and the history?

Well, I guess it’s because I like to talk about evolution. I am a firm believer in Darwinian evolution and I see many parallels in the evolution of computers and software. We are all here together in this room not only because we share the DNA of a common ancestor but also because our chosen programming languages such as, Clipper, Visual Foxpro and Xbase++ all share the DNA of a common language ancestor. There are similarities to Darwinian “natural selection” in that programming languages also die out or survive based on similar principles except that in the world of computing there are forces that can cause good software to go extinct and bad software to thrive but, for the most part, they too must adhere to the rule of “survival of the fittest”. In evolution terms, survival is based on which species can best adapt to a changing physical environment and the evolutionary "arms race". In software terms, survival is based on which language can best adapt to new platforms and new ideas through selective mutation.

In 1995, our world started changing in ways that challenged even the most robust languages as Microsoft introduced us to Windows and GUI. dBase failed to survive through this transition because it was bought by Borland and was treated like a step child. dBase-IV was a disaster in design and died a quick death because Borland was more interested in evolving their Delphi (pascal) language to Windows. Clipper failed to survive through this transition because it was bought by Computer Associates and their business model was to buy products and suck out the life’s blood. They attempted to move Clipper developers to a new product named Visual Objects, but it too was a disaster and also died a quick death. For several years, Clipper developers were wandering through the wilderness looking for a new home for their applications. Most Clipper apps were so large that the legacy code could not adapt to any language that did not have dBase in its DNA. There were many more attempts to help with the Clipper migration with products such as FiveWin and Clip4Win, while other dBase dialects such as QuickSilver, dBXL, Arago and Force simply went extinct. Some Clipper programmers saw no clear migration path for them so they abandoned the Xbase language entirely and started using Delphi. A few even chose Visual Basic. Very few Clipper developers migrated to FoxPro for a variety of reasons even though the products shared the same DNA. Most recognized the limitations of an interpretive language.

FoxPro survived through this transition because it was bought by Microsoft, and so they had an interest in making it work with Windows, even though it was not well integrated with their product line and did not support necessary features such as multi-threading and a compiler. Microsoft carried FoxPro until it no longer made any sense to support it and so it was announced that support will be terminated.

Now it is the year 2015. 20 years have gone by since the upheaval of 1995 that caused the extinction of so many dBase-dialect languages. 16 years have gone by since Alaska Software introduced the product Xbase++ as the natural successor to the dBase dialect language. There was a 4-year period in which Clipper developers had to find a new path. Those who stayed with Clipper during those transition years were rewarded with the introduction of Xbase++ in 1999, and Xbase++ continues to thrive today and even to evolve. The number of dBase, FoxPro and Clipper developers combined were likely in the hundreds of thousands whereas the Xbase++ developer community is many times smaller.

How is it possible that such a small company can still be developing and supporting the best dBase dialect language ever created while every other one went extinct? The answer lies in the philosophy of the development team. Alaska Software is a technology-driven company whereas all the large businesses that assimilated the now extinct languages are marketing-driven. In 1999 I bought Xbase++ and wrote my first "Hello World" programs in Xbase++; first in text mode (Figure 1) and then in GUI mode (Figure 2).

```
@ 1,1 SAY ‘Hello World’
```

Figure 1 – Text Based

```
#include “appevent.ch”

oDlg := XbpDialog():new(,,{300,300},{210,130},,.t.)
oDlg:taskList := .t.
oDlg:create()
oStatic := XbpStatic():new(oDlg:drawingArea,,{0,0},{200,100},,.t.)
oStatic:caption := ‘Hello World’
oStatic:create()
nEvent := 0

DO WHILE nEvent # xbeP_Close
  nEvent := AppEvent( @mp1, @mp2, @oXbp, 1 )
ENDDO
```

Figure 2 – GUI Based

Even though a GUI Hello World was more complex, it took very little time to convince me that Xbase++ was the rightful successor to Clipper. At that time, I had already had 17 years of experience with dBase and Clipper and had many Clipper customers who needed my help in moving to Xbase++ and GUI, and so I felt that it was necessary to help them make this migration as painless as possible. It became obvious that they needed a way to build complex GUI screens without first having to learn object-oriented programming, the GUI classes, and all the complexities of screen management. It occurred to me that I could write a language extension that would be familiar to them, would produce GUI screens as complex as any written in native Xbase++ and allow them to be 10-20 times more productive.

RAD Systems

From the early days of dBase, programmers have been trying to find faster and more reliable ways to develop applications. A plethora of third-party products were developed that promised they could reduce the amount of programming steps required to make an application. Some were screen generators that allowed a user to drag items around on a screen and would then write out the code. Some were data-driven systems that kept the bulk of the application in meta-data files which were used to either generate code or would use the meta-data during the running of the application to paint the screens and/or manage the opening of databases. Nearly all of these programs advertised themselves as RAD (Rapid Application Development) systems. I even wrote one of these systems myself for a customer. It was called DataMinder. I added a lot of features to my Clipper add-on product, dCLIP, to use as a library for this RAD system. In the late 80's and the 90's it seemed that this was the future of programming, but nearly all of these concepts have withered away and gone extinct for one reason or another.

I did several seminars in the 90's about data-driven programming. I was a believer. I wrote, or helped to write, several large applications that utilized data-driven programming techniques in Clipper. What I learned from that experience was that such programs are very difficult to modify and debug and that they have a tendency to be slow performers.

Another drawback was that a good data-driven engine is so complex that developers would spend so much time keeping the engine running that there was no time left for developing a good application. One of the applications that I helped to write was used by an Automobile Transmission Rebuilding company. The application had evolved into an enterprise-level application that handled every part of the operation including manufacturing, accounting and sales. It worked well for 10 years and then I was asked to help convert it to Windows.

I realized that given the concept of data-driven programming was already failing in the simplest of platforms, ie. text-based DOS, imagine the complexity of such a design for GUI application. I needed to think about a new RAD concept for GUI applications. I wanted to develop a GUI programming language, written entirely in Xbase++ that would allow developers with a lot of Clipper or FoxPro 2.6 experience to easily adapt to this new paradigm and become quickly productive. None of these developers had experience in object-oriented programming and none of them had been exposed to GUI controls. When they saw Xbase++ and looked at the sample code, it looked like a revolutionary language, not an evolutionary language. I wanted to change that image of Xbase++ and help my customers migrate. When other languages evolved from DOS to Windows, the developers solved this problem by creating a "workbench" type of environment that allowed the programmer to grab GUI controls from a toolbox, drag them to the screen, enter properties and then save the results as code. Alaska Software attempted to create a "form designer", but like many other form designers it had too many drawbacks and was rarely used in applications. Other 3rd party form designers also failed to deliver the goods. It was soon obvious to me that an entirely new approach was needed to solve this problem. My inspiration came from some experience with SQL (Structured Query Language). I had learned from experience that SQL statements were much more productive and powerful than equivalent ISAM code in which the programmer had to write complicated loops and/or relations, using filters, scopes, joins, seeks, indexes, etc. to achieve a resultant data set. Furthermore, the source code did not very well represent what it was that the programmer was trying to achieve without carefully interpreting each line of code.

SQL has been instrumental in simplifying data access in large applications and therefore has become predominant in most languages. This is because **SQL is a descriptive language**. SQL describes the desired dataset rather than coding the methods to achieve that result. It eliminates the need for all that complicated code because that code resides on the SQL server.

So I started putting together in my mind a new concept, which I will refer to now as **SGL (Structured GUI Language).** I wanted to make it possible to describe all the elements of a complicated GUI screen in a structured language that was easy to write and also to understand. It needed to have the ability to code the screens in a set of commands in which the command name described the type of GUI control and each command parameter described the properties of the control, including its relationship to all other controls. In Clipper, there was a similar concept that created @ .. SAY .. GETs on the screen and then managed the navigation through the screen via a "reader" command named READ. SGL is an Xbase++ adaptation of a **Domain-Specific Language (DSL)** . It occurred to me at this approach could be applied to every possible kind of GUI control. The advantages would be numerous.

1. It would make it easy to convert existing Clipper text-based code to Xbase++ GUI-based code.

2. It would eliminate the need for Clipper programmers to learn object-oriented programming and Xbase parts, in the same way that SQL programming eliminates the need to learn all the db*() database functions, except when it comes to GUI there are many more controls and properties to learn.

3. A structured GUI language would dramatically simplify the design of complicated GUI screens and make it unnecessary to use a form designer.

4. The commands are Xbase++ code therefore they are part of the .PRG source and can be combined with the business logic. This makes the code much easier to read and maintain than disjointed concepts like form designers.

5. Because the commands are Xbase++ code, they can include debugging information entered by the programmer.

6. The programmer can easily visualize the screen design and the relationships between GUI controls by just looking at the source code.

7. The programmer can write code in a syntax that is very forgiving and does not require knowing the order of parameters or the names of methods and iVars.

I knew that everything had to be written entirely in Xbase++ for this concept to become real and I also knew that Xbase++ supported the very features necessary to make this happen.

1. A powerful pre-processor capable of creating commands with numerous optional clauses.

2. The ability to bind LOCAL variables and Instance variables to the naming of GUI objects for the creation of parent/child relationships and linking the data to the GUI object.

3. A multi-dimensional, ragged array system that served as a container for every GUI control, every property, every parent/child relationship and every callback.

4. The ability to create new classes that inherited from Xbase Parts and other classes which will be used by the reader to create the screens from the definitions in the array.

5. The ability to create a reader function which will build and render the screens from the information in the array.

6. The ability to control events in the reader for screen management and navigation.

7. The ability to give the programmer the option of using pixel-based coordinates or text-based coordinates similar to Clipper.

8. The ability to give the programmer the option of using a top-down coordinate system or a bottom-up coordinate system.

Declarative vs Imperative Programming

I have used both **declarative** and **imperative** programming techniques over the years and I have come to find that declarative coding not only creates faster development but also easier maintainability because it is always easier to read and understand the source code. Some programmers prefer to call it **linear** vs **lateral** programming. The differences in terminology are minor and therefore irrelevant to this subject matter.

In a declarative approach, the application screen design and the business logic are usually in the same source file. Maintaining the code seldom requires opening more than one file and the programming can all be done within the same editing program. Applications are usually broken down into modules in which the source for an entire module may exist in only a few functions or class methods. A module is usually a program that encapsulates all the functionality for a given task or database manager. For example, a module may be a "customer" screen which provides for browsing, searching and data-entry of customers. Because all tasks supporting customer management are related, it makes sense that all the code should be in a single source file whenever practical. In a linear programming style, the code that designs the screen elements should be placed in a top-down structure that easily defines the relationships between all the GUI controls. The more function calls or class methods that are required to do this, the more disjointed is the code and the harder it is to follow the logic and the structure.

Languages in which the screen design is not done in code at all requires many more editing tools, and such tools do not lend themselves to any concept of linear programming. On the contrary, this is a more lateral style and when taken to an extreme can even be referred to as "atomic." Visual Objects was an example of a programming language that had everything in a repository with each VARIABLE, FUNCTION, COMMAND, CLASS, METHOD, iVAR store as an "atom".

Visual Objects failed for many reasons but, IMO, a big reason was very difficult for a programmer to use his/her own editor for writing a program. Data driven programs have the same set of problems in which much of the code is not code at all but instead a set of records in a database that defines screen output. Alaska Software has been struggling for years trying to adapt Xbase++ to the needs of the VFP community because Visual FoxPro developers could only create screens from a design tool and not from code. In the early days of Xbase++, Alaska Software decided that the language was more important than the "development environment". The language was so powerful that each programmer could create their own development environment. I used the Xbase++ language to create my own debugging tools, a dot-prompt interpreter, and a declarative GUI language. Clipper programmers did not complain because they were used to writing code to accomplish a given task. Visual FoxPro programmers come from the Microsoft world in which they were given everything needed to create a program without having to embrace the GUI elements of the language, therefore Alaska Software is finding themselves in the unenviable position of having to create an entirely new programming environment even though it will have absolutely no effect on producing a better end application.

Bobby Drakos and I have been talking about this for years and have been wondering why Visual Foxpro developers have been so slow at jumping aboard whereas Clipper developers were instant believers and their applications are the proof. I have been to every SWFox/SWXbase conference in the past 5 years and have seen no further movement towards Xbase++ even though Visual Foxpro applications have no other language in which to migrate. So we thought we would make an effort to expose Visual FoxPro developers to an idea that they can use right now and not wait for Polar Fox or some other environment.

An SGL Architecture

Earlier in this session I said that I like to talk about evolution and so here is a discussion of the evolution of a Structured GUI Language and why I finally ended up with the technique I created in eXpress++. I have seen other techniques that programmers have used to create their own command language for GUI, and although they can handle simplified GUI design, they fall far short when it is necessary to create complex and robust applications. In the following examples it will become clear why more levels of abstraction are needed for a fully evolved system.

I have seen some resistance to the SGL idea from seasoned programmers because they are transfixed on the idea that this isn't really programming and that a command-based language for GUI is like going backward rather than going forward. This is an illogical argument in the same way that programmers who refuse to use SQL say that only ISAM can handle the complexities of a robust database browsing system. I too had that opinion for many years but I evolved to understand why SQL became more popular over the years and ISAM programming became less popular. ISAM was not replaced by SQL. On the contrary, SQL engines are all driven by ISAM navigation and are mostly written in function based or object-oriented languages. The idea of SQL came about because it was taking too long for application programmers to get the datasets they needed. They had to have a language that could improve their productivity and give them reliable results. SGL solves the same problem for GUI screen design that SQL solves for database management.

Before starting this project, I had to write down some objectives to insure that this would not soon become another extinct idea.

1. Easy adaption to existing application code for migration purposes.

2. Good performance.

3. Must be written entirely in Xbase++.

4. Must be able to evolve in features and robustness.

5. Must be able to create applications as complex and robust as those written entirely in object-oriented or function based code.

6. Must increase programmer productivity 10 to 20 times over native language programming even when writing entirely new applications.

7. Must be a "language extension" rather than a "framework". Applications written in the SGL syntax must not have a specific look or functionality. Flexibility is an absolute must.

8. Must have no restrictions in the use of LOCAL, PRIVATE, STATIC and PUBLIC variables. Should allow application code that uses only LOCAL variables. Must NOT REQUIRE any PUBLIC variables or PUBLIC functions to bind GUI elements or other GUI screens to each other.

9. Must have no restrictions in the development of multi-threaded applications.

10. Must allow the ability to create any kind of GUI screen or relationships to other GUI screens following all the rules and capabilities of Xbase Parts including Parent, Owner, Sibling relationships and screen modality.

11. Must be compatible with screens and business logic written using native language programming, ie SGL objects can be intermingled with all other Xbase++ GUI objects.

12. GUI classes used by the SGL language must be Xbase Parts or derived from Xbase Parts and respond to the same methods and iVars.

13. Must offer features that are difficult to develop using other methodologies.

14. Installation must have a directory structure that makes it simple to deploy a patch for fixing bugs or adding a feature without needing to wait for a new release.

15. An application that uses the SGL language must be allowed to be any common programming structure, be it function based or object oriented, ie. SGL commands should be usable within OOPS methods or functions.

16. The SGL language must have a common structure for all GUI controls. For example subclauses like WHEN, HIDE, FONT, COLOR, RESIZE, SIZE, LOCATION, PARENT, VALID, GOTFOCUS, LOSTFOCUS, DRAG, DROP, TOOLTIP, MESSAGE, SUBCLASS, etc. must have the same meaning for each GUI command.

17. The programmer must be able to SUBCLASS a GUI command to build custom features into the underlying class architecture for each command.

18. The SGL language commands must not conflict with or replace any native Xbase++ commands, classes or functions.

19. The SGL language must allow flexibility in handling of data types that are bound to GUI controls. For example, a color may be passed as a numeric value, a RGB array or a code block.

20. The underlying architecture must also handle all the screen management and event management such as navigation rules, resizing rules, scaling, drag and drop, etc.

21. Must be well documented and include a plethora of examples.

The Xbase++ Pre-processor

In my opinion, the pre-processor gets the "most valuable feature" award when it comes to programmer productivity. This is because it provides the abstraction concepts necessary to reduce the number of lines of code needed to create enterprise-level applications.

Look at this code example (Figure 3):

```
SELECT INTO BOISE NAME,PHONE WHERE ZIP="83706" ORDER BY PHONE
```

Figure 3

This is a classic SQL SELECT statement which simplifies a Query-based method of data access and is why SQL is so popular a language. Due to the fact that compiled languages really do not understand commands such as this, the elements must be parsed into something that can be understood, and so there must be a parser that would convert the command to a set of ordered parameters that can be passed to a function or an object. The SQL function input parameters may look something like this (Figure 4):

```
SELECT INTO BOISE NAME,PHONE WHERE ZIP="83706" ORDER BY PHONE
```

Figure 4

```
SQLSelect( aColumns, cWhere, cOrderBy, cInto )
```

Figure 4a

If you were given the task of writing an SQL Query Parser you could use the Xbase++ pre-processor to accomplish this task (Figure 5).

```
#command SELECT [INTO <into>] <columns,...> ;
        [WHERE <where>] [ORDER BY <order>]; => ;
     SQLSelect( {<(columns)>}, <(where)>, <(order)>, <(into)> )
```

Figure 5

The above SELECT statement (Figure 5) would be converted to the following by the pre-processor during the compile process:

```
SQLSelect( {"NAME", "PHONE"}, 'ZIP="83706"', "PHONE", "BOISE" )
```

Figure 6

Most Xbase++ commands are defined in STD.CH. Look at this file (Figure 6) to get a better understanding of how the pre-processor is used to create the Xbase++ command set.

To create a set of SGL commands, we would need to design a GUI control syntax in which all the commands would be placed in a single source file, eg SGL.CH.

A Simple design

The simplest kind of SGL system is one in which only a CH file is created to define the commands. In the below example there are 2 commands; a command for the DIALOG window and a command for SAY..GET. This demonstrates a need that most Clipper programmers had when converting their applications. SGL.CH contains the commands and SGL.PRG is the program that uses the commands. In this sample, there is only 1 level of abstraction whereby commands are converted directly into object-oriented code that uses the Xbase Parts GUI classes. The limitations are quite obvious in that this adaption doesn't incorporate any text-based coordinates, utilizes a bottom-up instead of a top-down coordinate system and doesn't handle picture clauses, validations keyboard navigation or when clauses. Adding such features to this kind of design is basically impossible or so complex as to be impractical.

```
#command @ <nRow> ,<nCol> DIALOG                                        ;
                [COLOR <nColor>]                                        ;
                [PARENT <oParent>]                                      ;
                [SIZE <nWidth>,<nHeight>]                               ;
                [OBJECT <oObject>]                                      ;
                [TITLE <cTitle>]                                        ;
  => ;
   [<oObject> :=] oDlg := XbpDialog():new( <oParent>,,{<nCol>,<nRow>}, ;
                [{<nWidth>,<nHeight>}],,.t.) ;
   ;[oDlg:title := <cTitle>] ;
   ;[oDlg:drawingArea:colorBG := <nColor>] ;
   ;oDlg:create();

#command @ <nRow> ,<nCol> SAY <cSay> GET <cVar>                         ;
                [SAYFONT <cSayFont>]                                    ;
                [GETFONT <cGetFont>]                                    ;
                [SAYCOLOR <nSayFg> [,<nSayBg>] ]                        ;
                [GETCOLOR <nGetFg> [,<nGetBg>] ]                        ;
                [PARENT <oParent>]                                      ;
                [SAYSIZE <nSayWidth> [,<nSayHeight>]]                   ;
                [GETSIZE <nGetWidth> [,<nGetHeight>]]                   ;
                [SAYOBJECT> <oSayObject>]                               ;
                [GETOBJECT> <oGetObject>]                               ;
  => ;
   [<oSayObject> :=] oXbp := XbpStatic():new(oDlg:drawingArea,,{<nCol>,<nRow>},,,.t.);
   ;[oXbp:setFontCompoundName(<cSayFont>)] ;
   ;oXbp:autoSize := .t. ;
   ;oXbp:caption := <cSay> ;
   ;oXbp:create();
   ;[oXbp:setSize({<nSayWidth>,<nSayHeight>})] ;
   ;[oXbp:setColorFG(<nSayFg>)] ;
   ;[oXbp:setColorBG(<nSayBg>)] ;
    [<oGetObject> :=] oXbp :=
      XbpSle():new(oDlg:drawingArea,,{<nCol>+oXbp:currentSize()\[1]+10,<nRow>},,,.t.);
   ;oXbp:setFontCompoundName('10.Lucida Console') ;
   ;[oXbp:setFontCompoundName(<cGetFont>)] ;
   ;oXbp:dataLink := {|u|iif(u==nil,<cVar>,<cVar>:=u)};
   ;oXbp:autoSize := .t. ;
   ;oXbp:tabStop := .t. ;
   ;oXbp:bufferLength := 1000 ;
   ;oXbp:create();
   ;oXbp:setData();
   ;[oXbp:setSize({<nGetWidth>,<nGetHeight>})] ;
   ;[oXbp:setColorFG(<nGetFg>)] ;
   ;[oXbp:setColorBG(<nGetBg>)] ;
   ;oXbp:killInputFocus := {|a,b,o|o:getData()} ;
```

Figure 7 - SGL1.CH - The command definition file

```
#INCLUDE "sgl1.CH"
#INCLUDE "gra.CH"
#INCLUDE "appevent.CH"

FUNCTION Main()

LOCAL oXbp, oDlg, cName, cAddress, cCity, cState, cZip, nEvent, mp1, mp2

cName := Space(25)
cAddress := Space(25)
cCity := Space(25)
cState := 'CA'
cZip := Space(5)

@ 350,250 DIALOG SIZE 370,170 TITLE 'My Dialog' COLOR GRA_CLR_CYAN

@ 100,10 SAY 'Name' GET cName SAYSIZE 60,20
@  80,10 SAY 'Address' GET cAddress SAYSIZE 60,20
@  60,10 SAY 'City' GET cCity SAYSIZE 60,20
@  40,10 SAY 'State' GET cState SAYSIZE 60,20
@  20,10 SAY 'Zip' GET cZip SAYSIZE 60,20

nEvent := 0
DO WHILE nEvent # xbeP_Close
  nEvent := AppEvent(@mp1,@mp2,@oXbp,1)
  IF Valtype(oXbp) == 'O' .AND. nEvent > 0
    oXbp:handleEvent(nEvent,mp1,mp2)
  ENDIF
ENDDO

RETURN nil
```

Figure 7a - SGL.PRG – The application

A Better Design - Architecture is Everything

To insure that a design concept does not go extinct, the architecture must support the ability to grow major new features without degrading performance or reliability.

A more robust design is one in which a CH file is used to define the commands and a reusable READER function is used to render the screen. In this sample, there are 2 levels of abstraction. The first level converts commands to properties which are added to an array. The second level passes the array of definitions to a Reader function and creates the screen. Each array element defines a GUI control and each array sub-element defines a property of the GUI control. This system is based on Xbase Parts for the GUI classes and each Xbase Part contains a set of iVars and methods that are "standardized", thereby making it even easier to create a SGL architecture. For example, all GUI controls set the location, size, font, colors, presentation, etc. in the same way therefore the array structure should be designed around this commonality. In this scenario, notice that there is less code required in the source that uses the commands due to the fact that there are less local variables needed and the reader also handles the events.

```
#define SGL_TYPE        1
#define SGL_SUBTYPE     2
#define SGL_ROW         3
#define SGL_COL         4
#define SGL_COLOR       5
#define SGL_SIZE        6
#define SGL_CAPTION     7
#define SGL_FONT        8
#define SGL_PICTURE     9
#define SGL_OBJECT     10
#define SGL_PARENT     11
#define SGL_DATALINK   12
#define SGL_ACTION     13
#define SGL_OPTIONS    14

#define SGL_ARRAYSIZE  14

#define SGL_TYPE_DIALOG    1
#define SGL_TYPE_STATIC    2
#define SGL_TYPE_GET       3
#define SGL_TYPE_BUTTON    4
```

Figure 8 – SGL2.CH

```
#command READ => SGLRead( aSGL )

#command @ <nRow> ,<nCol> SGLDIALOG                                         ;
                [COLOR <nColor>]                                            ;
                [PARENT <oParent>]                                          ;
                [SIZE <nWidth>,<nHeight>]                                   ;
                [OBJECT <oObject>]                                          ;
                [TITLE <cTitle>]                                            ;
  => ;
   AAdd( aSGL, Array(SGL_ARRAYSIZE) )                                       ;
   ;SGLSet(aSGL,SGL_TYPE,SGL_TYPE_DIALOG)                                   ;
   ;SGLSet(aSGL,SGL_ROW,<nRow>)                                             ;
   ;SGLSet(aSGL,SGL_COL,<nCol>)                                             ;
   ;[SGLSet(aSGL,SGL_COLOR,{nil,<nColor>})]                                 ;
   ;[SGLSet(aSGL,SGL_SIZE,{<nWidth>,<nHeight>})]                            ;
   ;[SGLSet(aSGL,SGL_CAPTION,<cTitle>)]                                     ;
   ;[SGLSet(aSGL,SGL_PARENT,AnchorCB(@<oParent>))]                          ;
   ;[SGLSet(aSGL,SGL_OBJECT,AnchorCB(@<oObject>))]                          ;


#command @ <nRow> ,<nCol> SGLSTATIC                                         ;
                [CAPTION <cCaption>]                                        ;
                [FONT <cFont>]                                              ;
                [COLOR <nFg> [,<nBg>] ]                                     ;
                [PARENT <oParent>]                                          ;
                [SIZE <nWidth> [,<nHeight>]]                                ;
                [TYPE <nSubType>]                                           ;
                [OBJECT <oObject>]                                          ;
                [OPTIONS <nOptions>]                                        ;
  => ;
   AAdd( aSGL, Array(SGL_ARRAYSIZE) )                                       ;
   ;SGLSet(aSGL,SGL_TYPE,SGL_TYPE_STATIC)                                   ;
   ;SGLSet(aSGL,SGL_ROW,<nRow>)                                             ;
   ;SGLSet(aSGL,SGL_COL,<nCol>)                                             ;
   ;[SGLSet(aSGL,SGL_COLOR,{<nFg>,<nBg>})]                                  ;
   ;[SGLSet(aSGL,SGL_SUBTYPE,<nSubType>)]                                   ;
   ;[SGLSet(aSGL,SGL_SIZE,{<nWidth>,<nHeight>})]                            ;
   ;[SGLSet(aSGL,SGL_CAPTION,<cCaption>)]                                   ;
   ;[SGLSet(aSGL,SGL_FONT,<cFont>)]                                         ;
   ;[SGLSet(aSGL,SGL_PARENT,AnchorCB(@<oParent>))]                          ;
   ;[SGLSet(aSGL,SGL_OBJECT,AnchorCB(@<oObject>))]                          ;
```

Figure 8a – SGL2.CH

```
#command @ <nRow> ,<nCol> SGLSAY <cSay> GET <cVar>                           ;
                [SAYFONT <cSayFont>]                                         ;
                [GETFONT <cGetFont>]                                         ;
                [SAYCOLOR <nSayFg> [,<nSayBg>] ]                             ;
                [GETCOLOR <nGetFg> [,<nGetBg>] ]                             ;
                [PARENT <oParent>]                                           ;
                [SAYSIZE <nSayWidth> [,<nSayHeight>]]                        ;
                [GETSIZE <nGetWidth> [,<nGetHeight>]]                        ;
                [SAYOBJECT> @<oSayObject>]                                   ;
                [GETOBJECT> @<oGetObject>]                                   ;
  => ;
   AAdd( aSGL, Array(SGL_ARRAYSIZE) )                                        ;
   ;SGLSet(aSGL,SGL_TYPE,SGL_TYPE_STATIC)                                    ;
   ;SGLSet(aSGL,SGL_ROW,<nRow>)                                              ;
   ;SGLSet(aSGL,SGL_COL,<nCol>)                                              ;
   ;[SGLSet(aSGL,SGL_COLOR,{<nSayFg>,<nSayBg>})]                             ;
   ;[SGLSet(aSGL,SGL_SIZE,{<nSayWidth>,<nSayHeight>})]                       ;
   ;[SGLSet(aSGL,SGL_CAPTION,<cSay>)]                                        ;
   ;[SGLSet(aSGL,SGL_FONT,<cSayFont>)]                                       ;
   ;[SGLSet(aSGL,SGL_PARENT,AnchorCB(@<oParent>))]                           ;
   ;[SGLSet(aSGL,SGL_OBJECT,AnchorCB(@<oSayObject>))]                        ;
   AAdd( aSGL, Array(SGL_ARRAYSIZE) )                                        ;
   ;SGLSet(aSGL,SGL_TYPE,SGL_TYPE_GET)                                       ;
   ;SGLSet(aSGL,SGL_DATALINK,AnchorCB(@<cVar>))                              ;
   ;[SGLSet(aSGL,SGL_COLOR,{<nGetFg>,<nGetBg>})]                             ;
   ;[SGLSet(aSGL,SGL_SIZE,{<nGetWidth>,<nGetHeight>})]                       ;
   ;[SGLSet(aSGL,SGL_FONT,<cGetFont>)]                                       ;
   ;[SGLSet(aSGL,SGL_PARENT,AnchorCB(@<oParent>))]                           ;
   ;[SGLSet(aSGL,SGL_OBJECT,AnchorCB(@<oGetObject>))]                        ;

#command @ <nRow> ,<nCol> SGLBUTTON                                          ;
                [CAPTION <cCaption>]                                         ;
                [FONT <cFont>]                                               ;
                [COLOR <nFg> [,<nBg>] ]                                      ;
                [PARENT <oParent>]                                           ;
                [ACTION <bAction>]                                           ;
                [SIZE <nWidth> [,<nHeight>]]                                 ;
                [OBJECT <oObject>]                                           ;
  => ;
   AAdd( aSGL, Array(SGL_ARRAYSIZE) )                                        ;
   ;SGLSet(aSGL,SGL_TYPE,SGL_TYPE_BUTTON)                                    ;
   ;SGLSet(aSGL,SGL_ROW,<nRow>)                                              ;
   ;SGLSet(aSGL,SGL_COL,<nCol>)                                              ;
   ;[SGLSet(aSGL,SGL_COLOR,{<nFg>,<nBg>})]                                   ;
   ;[SGLSet(aSGL,SGL_SIZE,{<nWidth>,<nHeight>})]                             ;
   ;[SGLSet(aSGL,SGL_CAPTION,<cCaption>)]                                    ;
   ;[SGLSet(aSGL,SGL_FONT,<cFont>)]                                          ;
   ;[SGLSet(aSGL,SGL_ACTION,<bAction>)]                                      ;
   ;[SGLSet(aSGL,SGL_PARENT,AnchorCB(@<oParent>))]                           ;
   ;[SGLSet(aSGL,SGL_OBJECT,AnchorCB(@<oObject>))]                           ;
```

Figure 8b – SGL2.CH

```
#INCLUDE "sgl2.CH"

FUNCTION Main()

LOCAL aSGL[0], cName, cAddress, cCity, cState, cZip, oStatic

cName := Space(25)
cAddress := Space(25)
cCity := Space(25)
cState := 'CA'
cZip := Space(5)

@ 350,250 SGLDIALOG SIZE 370,170 TITLE 'My Dialog' COLOR GRA_CLR_CYAN

@ 10,10 SGLSTATIC SIZE 330,150 OBJECT oStatic TYPE XBPSTATIC_TYPE_GROUPBOX ;
        CAPTION 'My Address Info'

@ 100,10 SGLSAY 'Name' GET cName SAYSIZE 60,20 PARENT oStatic
@  80,10 SGLSAY 'Address' GET cAddress SAYSIZE 60,20 PARENT oStatic
@  60,10 SGLSAY 'City' GET cCity SAYSIZE 60,20 PARENT oStatic
@  40,10 SGLSAY 'State' GET cState SAYSIZE 60,20 PARENT oStatic
@  20,10 SGLSAY 'Zip' GET cZip SAYSIZE 60,20 PARENT oStatic

READ

RETURN nil

// SGLREAD.PRG

#INCLUDE "SGL2.CH"
#INCLUDE "appevent.CH"

FUNCTION SGLRead( aSGL )

LOCAL oXbp, i, oParent, nRow, nCol, aColor, cFont, nEvent, ;
      mp1, mp2, aSize, oDlg, cCaption, bDataLink, xOptions, ;
      bObject, nSubType, bAction

FOR i := 1 TO Len(aSGL)

  oParent := aSGL[i,SGL_PARENT]
  IF Valtype(oParent) == 'B'
    oParent := Eval(oParent)
  ELSEIF Valtype(oDlg) == 'O'
    oParent := oDlg:drawingArea
  ENDIF

  nRow := aSGL[i,SGL_ROW]
  nCol := aSGL[i,SGL_COL]
  IF nRow == NIL .AND. nCol == nil .AND. Valtype(oXbp) == 'O'
    nRow := oXbp:currentPos()[2]
    nCol := oXbp:currentPos()[1] + oXbp:currentSize()[1] + 10
  ENDIF
```

Figure 8c – SGL2.PRG

```
aSize := aSGL[i,SGL_SIZE]
cFont := aSGL[i,SGL_FONT]
aColor := aSGL[i,SGL_COLOR]
cCaption := aSGL[i,SGL_CAPTION]
bDataLink := aSGL[i,SGL_DATALINK]
xOptions := aSGL[i,SGL_OPTIONS]
bObject := aSGL[i,SGL_OBJECT]
nSubType := aSGL[i,SGL_SUBTYPE]
bAction := aSGL[i,SGL_ACTION]
oXbp := nil

IF aSGL[i,SGL_TYPE] == SGL_TYPE_DIALOG

  oDlg := XbpDialog():new( oParent,, {nCol,nRow},aSize,,.t.)
  oDlg:taskList := .t.
  oDlg:title := cCaption
  oDlg:create()
  IF !Empty(aColor)
    oDlg:drawingArea:setColorFG(aColor[1])
    oDlg:drawingArea:setColorBG(aColor[2])
  ENDIF

ELSEIF aSGL[i,SGL_TYPE] == SGL_TYPE_STATIC

  oXbp := XbpStatic():new( oParent,, {nCol,nRow},aSize,,.t.)
  oXbp:autoSize := .t.
  oXbp:caption := cCaption
  IF Valtype(nSubType) == 'N'
    oXbp:type := nSubType
  ENDIF
  IF Valtype(xOptions) == 'N'
    oXbp:options := xOptions
  ENDIF

ELSEIF aSGL[i,SGL_TYPE] == SGL_TYPE_GET

  oXbp := XbpSle():new( oParent,, {nCol,nRow},aSize,,.t.)
  oXbp:autoSize := .t.
  oXbp:dataLink := bDataLink
  oXbp:bufferLength := 500
  oXbp:killInputFocus := {|a,b,o|o:getData()}
  IF Empty(cFont)
    cFont := '10.Lucida Console'
  ENDIF

ELSEIF aSGL[i,SGL_TYPE] == SGL_TYPE_BUTTON

  oXbp := XbpPushButton():new( oParent,, {nCol,nRow},aSize,,.t.)
  oXbp:activate := bAction
  oXbp:caption := cCaption

ENDIF
IF oXbp == nil
  LOOP
ENDIF
```

Figure 8d – SGL2.PRG

```
  IF Valtype(cFont) == 'C'
    oXbp:setFontCompoundName(cFont)
  ENDIF

  oXbp:create()

  IF Valtype(aSize) == 'A'
    oXbp:setSize(aSize)
  ENDIF
  IF Valtype(aColor) == 'A'
    oXbp:setColorFG(aColor[1])
    oXbp:setColorBG(aColor[2])
  ENDIF
  IF Valtype(bObject) == 'B'
    Eval(bObject,oXbp)
  ENDIF

NEXT

nEvent := 0
DO WHILE nEvent # xbeP_Close
  nEvent := AppEvent(@mp1,@mp2,@oXbp,1)
  IF Valtype(oXbp) == 'O' .AND. nEvent > 0
    oXbp:handleEvent(nEvent,mp1,mp2)
  ENDIF
ENDDO

RETURN nil

* ----------

FUNCTION SGLSet( aSGL, nWhich, xValue )
RETURN ATail( aSGL )[ nWhich ] := xValue
```

Figure 8e – SGL2.PRG

Handling of Variables in an SGL Architecture

Two of the important features of Xbase++ are its ability to create pass-by-reference LOCAL variables and "detached LOCALs". Without this capability, an array-based SGL architecture would be a design disaster. This is referred to as "anchoring" a variable via a codeblock and is accomplished with the AnchorCB() function. AnchorCB() creates a Get-Set code block that can be evaluated at a later time, ie, in the GUIRead() function. Local variables may be passed by reference, thereby making it possible to have command clauses such as OBJECT, PARENT or VAR to bind the command to a local variable or an instance variable. Robust applications must employ all local variables to insure that variables are protected from calls to other functions or procedures.

The next step - eXpress++

A robust SGL architecture must be able to handle a much higher complexity than that shown earlier. This was an exercise in showing how the correct architectural design could evolve into something that could meet all the objectives for the SGL language I had in mind. What I had in mind became my product eXpress++ and it has been evolving now for 16 years along with hundreds of applications based on this concept.

In the next session, I will discuss eXpress++ and how to use it to create applications from scratch, or migrate existing Clipper or Visual FoxPro applications. Attendees of this session will receive eXpress++ and a 1 year subscription.

There are no strings attached. This is the full package including source code.

There is **no** white paper that will cover the subject matter in the afternoon session. Attendees will receive everything on a USB drive. It will include software, sample programs, documentation, MobileME (Mobile Multi-edit) and a trial version of Xbase++ 2.0 for those who do not already own Xbase++.