Java and FlagShip

"A prosper future is made of promoting communities like Linux community. Here it is one more a seed..."

I should admit before hand that I appreciate very much Flagship language. Even though Flagship was born with some advanced features, for instance: objects, inline C, etc. it still has lacking of some attributes that distinguish modern languages. It is not the time to expose those missing features; it is a computer language that allowed us using old Clipper source code and in many ways it stretched the lifetime of several of our systems. Recently Visual Flagship was put on the market with the main goal of creating visual user interfaces. For those people who tried it have realized that in fact visual user interfaces could be created, but far from the beauty and sophistication of others languages. So, how to create a true visual user interface – like those produced by Delphi or VisualBasic, etc. – and mix it with FS code? Even better, is that possible?

The answer is made up of four letters: J-A-V-A. Java is, in my opinion, one of best languages ever created and I intend to prove it by integrating Java code with Flagship code. Java is not simply a common computer programming language, it encloses a revolutionary philosophy about how we should create systems and this is why big companies such as IBM, HP, Sun, etc. use it nowadays.

Before we go any further, please, have a look at **Picture 1.** What do you think? The interface was build using Netbeans running with SkinLF, however, sometimes I use JBuilder or the simple and powerful 'vi'. All tools mentioned above are 'freeware' or have a freeware version. In our case those tools were used only to specify and settle position of visual components on the 'frame'. That part of the job may be viciously pleasant. Lets split our mission in three parts. **First** we are going to make use of an advanced feature of Flagship: *Objects*. That resource was very well implemented inside FS and I made use of it for nothing better than using objects to contact another language strongly object oriented. **Second**, inline C; another gracious feature of FS that allows source code written in C language cohabit to FS code. Did I say C language? Are we not going to use Java? Yes. But lets think a little bit about the matter: Flagship doesn't direct understand Java, Java in its turn doesn't understand Flagship, however both implement a manner to understand C code. Thus, C shall be our intermediate language! Smart conclusion! Finally, on the **Third** part we will merge everything in order to produce a real effective user interface. By this time you should be wondering whether is better or not waiting a bit more for improvements on Visual Flagship. Read through the article and see by yourself.

Picture 1

Part One: Objects

Object programming is more clear and efficient than procedural programming. This section will be more understandable in case you reader already know about object oriented programming.

We are going to use elementary concepts of the object oriented world, such as methods and attributes. Clarifying step-by-step the whole subject would be an impossibility since this article would assume a forbidden extension for publication. **Picture 2**, declares an FS object.

Do not bother about those colors. They help on referencing only. Say we are paying homage to 'vi'. For didactics purposes we may assume that in green color are the reserved words responsible for begin a section. In blue, words that specify data types. In red, the owner class of a section, and finally in orange object variables declarers. Others parts are use commonly on FS coding. That is the basic structure of a FS object. For details, please do read OBJ section of Flagship manual. There are many prime code excerpts inside section CMD:CLASS, INSTANCE – I would read everything in case I would care about using objects seriously.

Part Two: inline C

Inline C consists in allowing C code to be inserted directly into FS code. A detailed description is given on FS manual, section EXT under the title *Open C System API* – of course once more an agreeable reading for the weekend. Multisoft worked very well on this feature and thanks to that our 'mission' was in many ways much more easier. **Picture 3** shows an inline C code excerpt.

Looking at code on that picture we can correctly conclude that we are looking at our well-known C code. That source code may be merged directly into a FS application with no additional problems. On examples supplied by FS manual we realized that even FS variables can be reach from C code.

Picture 2

```
CLASS jvClass
  INSTANCE jv_clsname := "" AS CHARACTER

ACCESS jv_clsname CLASS jvClass AS CHARACTER
  Return jv_clsname

ASSIGN jv_clsname( vl ) CLASS jvClass
  jv_clsname := vl
  Return jv_clsname

METHOD Init( fg_nmcls ) CLASS jvClass
  // Todo código contido nesta seção será
  // executado na criação do objeto
  Return

METHOD isValidDate( dt ) CLASS jvClass
  Local bRet := .T.
  IF Empty(CtoD(dt))
    bRet := .F.
  Endif
  Return bRet

METHOD ... e assim por diante
        .
        .
```

Part Three: The Fusion

Java language have the ability of accessing native resources of the underlying systems. That can be accomplished using JNI (*Java Native Interface*). JNI on Linux is wrapped essentially on

Picture 3

```
#Cinline
{
  for( iCt = 1; iCt <= 10; iCt++ )
  {
    fprintf( stderr, ": %i : \r\n", iCt );
  }
}
#endCinline
```

dynamic libraries – commonly with .so file extension – that linked with certain application or library give to that application the power of accessing functions explicitly declared to that purpose. I used the word 'explicitly' because some small changes should be applied on functions declaration. **Picture 4** shows that changes.

Another important resource that JNI offers is the ability of calling a JVM (Java Virtual Machine) from non-Java applications, for instance, those written using Flagship. That was precisely what we intended initially, in another words, we are going to start a JVM from inside a FS app and put it at our disposal. So, lets make it work…

Even though it may look, at first glance, a complex coding it is not really, for things are made always in the same way. Looking at the code on **Picture 5** we realize that first thing the program does is create a virtual machine. This is obvious since the next functions have need of a JVM already initialized. On the following, we opened a DBF file then we create one index and 'RegFunc' is invoked. Do not search for that last function declaration because it is declared inside our mediator program written in C. That function registers some functions that shall be called from Java code. We create an object named 'ofrm' (implemented in 'jvrqjvm.prg') and the static method 'setUI' of Pesquisa class is called in order to activate SkinLF; finally our user interface is shown. From now on Java frame takes control of the application. In spite of executing a single record searching the combination of Java and FlagShip can be expanded to do much more useable things, such as: complex computation of data, DBF manipulation, very nice user interfaces, etc.

To accomplish that single searching task we have used some of the most advanced resources on both languages. So, why not using another advanced resource like UML (*Unified Modeling Language*) to show dependency relations inside our application? **Picture 6** shows that the item 'mediador.c' is the bridge between the languages. The remain items are not really new ones. In fact they play a common role on normal practice on each of the mentioned languages.

On the diagram and below of each item please

Picture 4

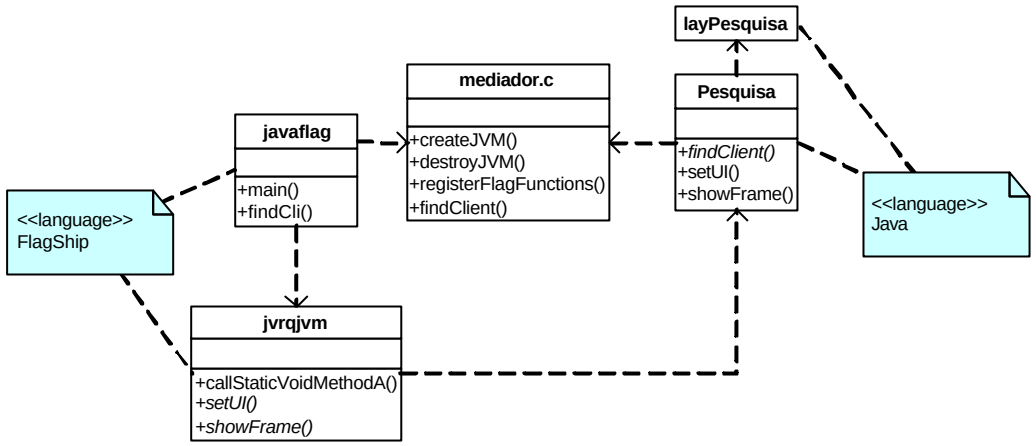
```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, object obj)
{
    printf("Hello World!\n");
    return;
}
```

pay attention to the allowed *operations*. Words on italic style indicate that even though that method exists it is not really implemented on that point. For instance, the real

Dependency Diagram

Picture 6



source code of *showFrame* is inside 'Pesquisa' class and it is not inside FS object 'jvrqjvm'. Despite of that we tolerate its appearance in order to get a precise abstraction. Another reason, obviously, is to demonstrate that that operation is available to anyone who creates instances of that class.

Together with the files of this article there is a Makefile file that specifies in which order the compilation process must happen. That order is crucial for the whole process. In another words, you will get messages errors whether you trying to compile 'javaflag.prg' before 'jvrqjvm.prg' since the former depends on the latter. However, 'javaflag.prg' do not depends on 'Pesquisa' Java class directly. So, the two compilation processes can exist at different times. The foretold situation means such an important fact: In case you want to update your Java classes you can do it independently of FS code. This do work in both directions. There is of course a constrain praying that if you try to invoke a non-existent method or class a runtime error will raise.

Well, here it is a way to put together Flagship and Java, both in peace and very together inside the same application. We did not solved all possible troubles, but we believe we set a light over the darkness in which this subject was inserted in, and we gave to those who want to start a nice beginning. "*Then, in the end, the beginning is not so hard.*"

Figura 5

```
#include "jvclasses.fh"

Function Main()
  IF jvmUpJVM()
    Use "clientes.dbf" Alias cli Shared New
    Index On cli->CODIGO to cli1
    Set Index to cli1
    RegFunc()
    Local ofrm := jvrqJVM{} AS jvrqJVM
    ofrm:setUI()
    ofrm:showFrame()
  Else
    Alert( "Erro ao carregar JVM" )
  Endif
  Return

Function findCli( codigo )
  Sele cli
  Go Top
  IF DbSeek(Padr(codigo, Len(&(IndexKey(0)))),.F.)
  Return { .T., cli->DESCRICAO }
  Endif
  Return { .F., "Erro na procura de registro" }
```

Requisites:

- Linux
- Flagship
- Java SDK 1.4

Sites:

JavaSDK	– *http://java.sun.com*
FlagShip	– *http://www.fship.com*
NetBeans	– *http://www.netbeans.org*
SkinLF	– *http://www.l2fprod.com*
Vim	– *http://www.vim.org*
JBuilder	– *http://www.borland.com/jbuilder*
UML	– *http://www.omg.org/uml*

Ricardo Delamar Roque - roque@gdysafety.com.br

First published on Revista do Linux #42 Jun./2003
www.revistadolinux.com.br (Brazilian publication)

The full source is available in .tgz format for a free download