Making Reusable Interface Components in Visual PRO/5

By Michael Martinez

The *Visual PRO/5 GUI Guide* offers the following advice to developers who need to upgrade their character-based applications to use graphical interfaces:

"..it is possible to gradually customize existing code for graphical interfaces step-by-step. Programs that should lend themselves easily to this type of migration include reports with separate pick-screen modules and *trapped* user input where external programs are called to handle exceptions. Update programs also usually contain pick-screens and *trapped* user input for exceptions."

The *GUI Guide* goes on to suggest that file maintenances may be more difficult to bring into the graphical world. Well, that may be true and it may *not* be true. This article discusses some modular design techniques that apply equally well to file maintenance screens as to report pick-screens and update pick-screens.

First let me explain some terms that you should become familiar with. For brevity's sake, however, I assume you are familiar with the concepts outlined in the *GUI Guide,* such as evaluation loops, contexts, and reading the event queue. Please refer to the *GUI Guide* for a full explanation of these concepts if you are not familiar with them.

Class A class is a set of things which share common operations and attributes. This definition should be familiar to anyone who has practiced object-oriented programming. You define objects that belong to given classes, and these objects inherit "properties" from their classes.

Modal A graphical object is "modal" if the operator using an application must do something with the object. Dialog boxes are often modal.

Non-Modal A graphical object is "non-modal" if the operator can ignore it. Think of a multi-document interface (MDI) application such as a word-processor where the operator is free to work in any of several windows at will. These are non-modal windows.

Parent A graphical window is a "parent" if it contains other graphical windows. The parent is the "frame" for the child windows and other graphical objects.

Resource A resource is a graphical object. A window or dialog is a resource, as is a button or other control placed in the window or dialog.

In Visual PRO/5, interfaces are built through graphical windows. These windows may contain menus, status bars, child windows, and various specialized controls such as buttons, boxes, and the like. You can develop design methodologies that use modular approaches to constructing these windows. Such methodologies may make use of the BASIS Resource Editor, or programs that create resources dynamically (i.e., at run-time).

Modular programming is not a new concept, but modular design for interfaces is an unusual concept for traditional Business BASIC applications. In following the examples of this article you'll adopt some modular design techniques to replace the traditional modular coding you may be familiar with in creating user interfaces. Each interface in a graphical application consists of various components, and our intention is to develop a methodology for reusing components as much as possible. One benefit of this approach will be a reduction in the amount of code you have to write for both new applications and applications being given a graphical upgrade.

Specifically, you will create reusable tool bars and windows. In order to be reusable, these interface components must contain only features common to a set or class of applications. Three such classes have already been mentioned: report pick-screens, update pick-screens, and file maintenance programs. Now file maintenance screens may not seem like they have much in common as each file is unique in some way. But file maintenance programs generally share common interface features

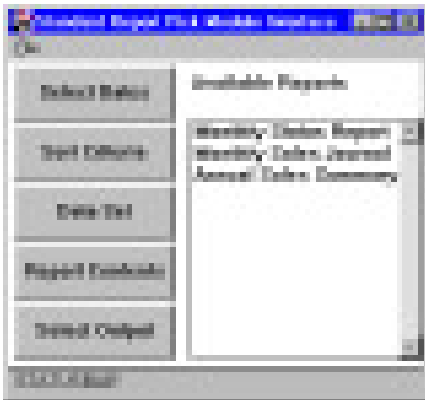
continued...

such as options to SAVE, DELETE, GET NEXT, etc.

The key to making reusable components is simply to identify the common interface features. For instance, will every report pick-screen be created in a window or dialog? A window may offer some advantages over a dialog, but whichever "base" is used is irrelevant. If a common style for pick-screens is adopted then they can all share resources. That is what this article is all about: sharing resources between as many programs as possible.

To illustrate this point, consider a class of report pick-screens that allow the operator to set date ranges, select output devices, establish sorting criteria and selection criteria, and determine output contents. You can design a "report launcher" program that uses a single master screen to present the user with choices about reports and options for each report. This launcher can be written in a variety of ways, and may use any number of interface designs.

There are five general properties that in the course of implementing will require specialized handling for each program, but nonetheless can be grouped to create a generic interface:



By default the report launcher will be notified if a push button, close box, or menu item is selected, but the launcher or the resource file will have to set the "Click or double-click on list item" event flag. For purposes of this article assume that this dialog is created in the BASIS Resource Editor and that the appropriate event flag is selected there.

The five buttons allow you to invoke custom dialogs (if need be) that will prompt the operator for the desired information, so that all reports can be launched from a single report manager. The list box allows you to provide the operator with a selective list of reports to be run. (This allows the application developer to customize report sets by user, department, module, application, and the like.)

Not all reports may require all five general properties. The buttons can be enabled/disabled based on items highlighted in the list box as the user clicks on each. Such flexibility would require more coding than simply making all five options available for every report regardless of its needs, but the additional work is not extensive.

For instance, if the report launcher uses a registry for all reports, it can quickly retrieve properties for each report in the form of Boolean switches. The template below illustrates a simple report registry layout:

Report_Property	**Template Name**
Report_Name	C(40)
Includes_Dates	U(1)
Includes_Sorts	U(1)
Includes_Data_Set	U(1)
Includes_Contents	U(1)
Includes_Output	U(1)

The report launcher contains an event processing loop that looks something like:

```
        sysgui=unt
        open (sysgui)"X0"
        report_list=unt
        open (report_list)"reports"
        dim event$:tmpl(sysgui)
        buffer_size=len(event$),
:       close_box$="X",
:       select_dates=101,
:       sort_criteria=102,
:       data_set=103,
:       report_contents=104,
:       select_output=105,
:       list_box=106

        while
                read record (sysgui,siz=buffer_size)event$
                if event.code$=close_box$ then
:                       break
                switch event.id
                case select_dates;
:                       call "datelist",data_data$,report_name$;
:                               break
```

continued...

```
            case sort_criteria;
:                       call "sortlist",sort_data$,report_name$;
:                       break
            case data_set
:                       call "select",select_data$,report_name$;
:                       break
            case report_contents;
:                       call "contents",field_list$,
:                              report_name$;
:                       break
            case select_output;
:                       call "output",output_dev$,report_name$;
:                       break
            case list_box;
:                       gosub list_box
            swend
      wend
      .
      .
      .
```

Note that the subroutine list_box would be the section of code where the report launcher enables and disables the buttons. This is based on the criteria it would retrieve from the list box by querying the list box for the current selected item:

```
      list_box:
            rem "You detected a click on the list box
            report_name$=ctrl(sysgui,event.id,1)
            dim report_property$:fattr(report_property$)
            find record (report_list,key=report_name$,
:                              dom=not_on_file)report_property$
            for i=select_dates to select_output;
:                 print (sysgui)'disable'(I);
:           next I
            report_name$=report_property.report_name$
            if report_property.includes_dates then
:                 print (sysgui)'enable'(select_dates)
            if report_property.includes_sorts then
:                 print (sysgui)'enable'(select_sorts)
            if report_property.includes_data_set then
:                 print (sysgui)'enable'(select_data)
            if report_property.includes_contents then
:                 print (sysgui)'enable'(select_contents)
            if report_property.includes_output then
:                 print (sysgui)'enable'(select_output)
      not_on_file:
      return
```

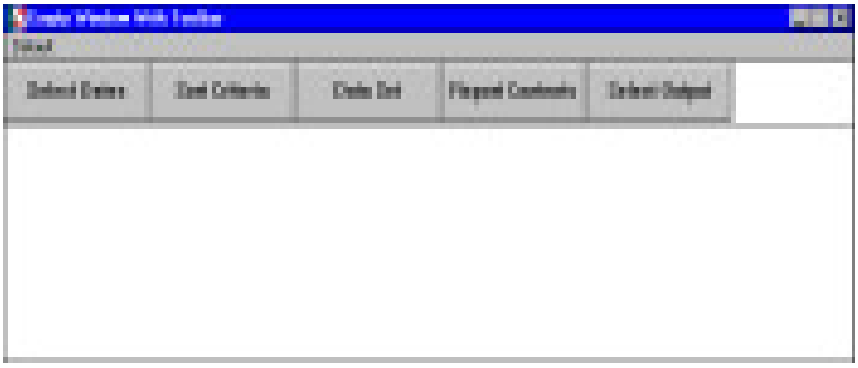
The subroutine “list_box” need not be a subroutine, but placing this block of code in a subroutine makes it easier to read the event loop logic.

The called programs may in turn call other programs which work with different modal dialogs. These dialogs should be modal so the report launcher does not have to filter out detail events. Let the overlays manage the complexities of validating date and sort ranges, field selections, and the like.

continued...

All the above is fine for a class of report programs that use the same general pick-screen logic, but what about pick-screens that require more detailed input? You cannot use the report launcher to handle those pick-screen options intuitively because it doesn't allow for extra input.

You can still make a reusable component, however, by writing a program that adds a child window to the current parent window in the form of a tool bar:



You'll put your five standard buttons on the tool bar and create some pick-screens that incorporate it into their designs.

The toolbar can be added to a resource dynamically so that you don't have to duplicate it across resource files. The advantage to using this program is that you can "dock" the toolbar to one side of the window, and perhaps offer customization capabilities to application interfaces at some future point by allowing operators to specify where they want to place toolbars.

The examples below call this program "toolbar", and it will accept as parameters two string variables, *operation*$ and *contents*$. The first string variable will be used to communicate with the program, and the second string variable will be used to retrieve data from "toolbar".

The operations that "toolbar" will perform are: *init*, *create*, *destroy*, *show*, *hide*, *move*, *enable*, and *disable*. You don't have to provide all of these features immediately. You can resort to "stub" programming that simply accepts certain calls and does nothing. Eventually, the "toolbar" may be extended to accommodate many more capabilities.

Unlike the reports handled by the launcher, these "customized" pick-screens will each require their own interface handler, a program that reads the event queue from the SYSGUI device and responds to those events. The shared resource among these pick-screens is the tool bar itself and the "toolbar" program that creates the "toolbar".

The first operation you'll pass to "toolbar" is "init". The program will assign a template to the *contents*$ variable so that the calling program can assign specific IDs for the child window and the controls that it will contain.

Next, you'll pass the "create" operation to "toolbar". This will tell the program to create the toolbar in the current context. Although each pick-screen handler will have to make reference to the fields in this template, they don't all need to have hard-coded assignments for creating the template. Let the shared resource do this.

So a call to "toolbar" will look like:

```
call "toolbar","create",contents$
```

continued...

Since nothing will be passed back in *operation*$ this is an internal string variable to "toolbar", and you can pass string literals to it.

"toolbar" itself, however, needs to do a little work with *operation*$. First, you should reformat the operation so that it can be evaluated:

```
rem
rem  Sample toolbar manager.
rem
rem  For Visual PRO/5
rem

     seterr exit_point

     enter operation$,contents$

     test$=fattr(contents$,err=assign_template)
     if operation$="init" then
:           goto assign_template
     goto evaluate_operation

     assign_template:

     dim contents$:"id:u(2),context:u(2),status:u(1),select_dates:u(2)"
:                 +",sort_criteria:u(2),data_set:u(2)"
:                 +",report_contents:u(2),select_output:u(2)"

     evaluate_operation:

     contents.status=0

     sysgui=unt
     open (sysgui)"X0"

     op_len=len(operation$)
     if op_len<7 then
:           operation$=operation$+fill(7-op_len)

     op_code=int((pos(operation$="create destroyshow hide
"
:                       +"enable disable",7)+6)/7)

     switch op_code
           case 1;
:                gosub create_toolbar;
:                break
           case 2;
:                print (sysgui)'context'(contents.context);
:                print (sysgui)'destroy'(0);
:                contents.status=1;
:                break
           case 3;
:                print (sysgui)'context'(contents.context);
:                print (sysgui)'show'(0);
:                contents.status=1;
:                break
```

continued...

```
            case 4;
:                   print (sysgui)'context'(contents.context);
:                   print (sysgui)'hide'(0);
:                   contents.status=1;
:                   break
            case 5;
:                   print (sysgui)'context'(contents.context);
:                   print (sysgui)'enable'(0);
:                   contents.status=1;
:                   break
            case 6;
:                   print (sysgui)'context'(contents.context);
:                   print (sysgui)'disable'(contents.id);
:                   contents.status=1
      swend

      exit_point:

      if sysgui then
:           close (sysgui)

      exit

      create_toolbar:
      dim fin_data$:tmpl(sysgui,ind=0)
      fin_data$=fin(sysgui,ind=0),
:     window_flag$=$00200810$,
:     child_context=fin_data.available_context,
:     contents.select_dates=101,
:     contents.sort_criteria=102,
:     contents.data_set=103,
:     contents.report_contents=104,
:     contents.select_output=105
      print (sysgui)'child'(contents.id,0,0,0,42,$$,
:           window_flag$,child_context)
      print (sysgui)'context'(child_context)
      print (sysgui)'button'(contents.select_dates,
:           0,0,120,40,"Select Dates",$$)
      print (sysgui)'button'(contents.sort_criteria,
:           120,0,120,40,"Sort Criteria",$$)
      print (sysgui)'button'(contents.data_set,
:           240,0,120,40,"Data Set",$$)
      print (sysgui)'button'(contents.report_contents,
:           360,0,120,40,"Report Contents",$$)
      print (sysgui)'button'(contents.select_output,
:           480,0,120,40,"Select Output",$$)
      contents.status=1,
:     contents.context=child_context
      return

      end
```

This program should work well with any program that needs to create the same type of resource. Note that the child window, though given its own context, is still a component of the parent window and thus must be assigned a unique control ID in the parent window's context.

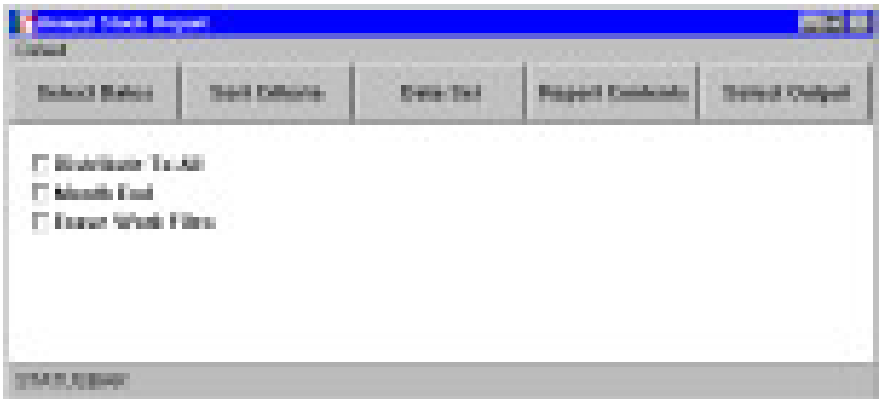
continued...

The calls to “toolbar” should look something like:

```
call "toolbar","init",contents$;
contents.id=10000
call "toolbar","create",contents$
call "toolbar","show",contents$
```

Note that this methodology will work equally well with TAOS procedures.

A third alternative for creating shareable resources is to design a resource file that creates the base window, the menu bar, the status bar, and a row of buttons across the top and/or side, and then copy that file to new names as you develop new pick-screens, such as the one below:



It should be apparent by now that what you can do for report pick-screens you can also do for file maintenances and update pick-screens as well. A typical file maintenance program may offer options such as deleting a record, getting the next record in sequence, getting the previous record, adding a new record, updating an altered record, etc. These generic functions lend themselves well to the toolbar concept, and it is not uncommon for graphical applications to employ multiple toolbars, especially when the design allows the operator to “customize” their environment by selecting toolbars that are “displayed”.

In the future I’ll examine some of the refinements which can be made to traditional applications so that they may take advantage of the graphical environment as much as possible.