Developing Principles of GUI Programming Using Views

Judith Bishop
Department of Computer Science
University of Pretoria, Pretoria 0002
South Africa
jbishop@cs.up.ac.za

Nigel Horspool
Department of Computer Science
University of Victoria, Victoria,
Canada V8W 3P6
nigelh@uvic.ca

ABSTRACT
This paper proposes that GUI development is as important as other aspects of programming, such as a sound understanding of control structures and object orientation. Far less attention has been paid to the programming structures for GUIs and certainly there are few cross language principles to aid the programmer. We propose that principles of GUIs can be extracted and learnt, and that they do enhance good programming practice. These principles have been implemented in our Views system which features an XML-based GUI description notation coupled with an engine that shields the programmer from much of the intricate complexity associated with events, listeners and handlers. The system is programmed primarily in C# for .NET, but is available in various forms for Java and for other platforms which support .NET through the SSCLI.

Categories and Subject Descriptors
D.3.2 C# I.7.2 XML, D.1.5 object oriented programming, D.1.0 General, D.3.3 Constructs and features, D.1.7 Visual programming.

General Terms - Languages

Keywords - graphical user interfaces, event-based programming, XML, platform independence.

1. INTRODUCTION
The purpose of this paper is to advance the state of the art of the development of graphical user interfaces (GUIs). In teaching programming, we like to stick to principles and to avoid, as much as possible, language specific and especially platform specific issues. The term *principles* is most often held to encompass control structures, data structures and object-orientation. However, for both introductory and industrial strength programs, input-output is as important a concept as a component and, in this day and age, input-output should definitely be GUI based.

Very little work has appeared on the principles of GUI development. By GUI development, we mean the manner in which a programmer realizes within a program the interaction (the I) that will take place with the user (the U) via graphical components (the G). Development follows from design. It should be stressed that we are not concerned here with design of GUIs – that is a whole area on its own and has been extensively studied by the HCI community.

Our premise is that, even for small programs, a GUI can be specified separately, and then linked in a coherent way to the rest of the computational logic of the program. The current practice is for GUIs to be specified by creating objects, calling methods to place them in the correct places in a window, and then linking them to code that will process any actions required. If hand-coded, such a process is tedious and error-prone. If a builder or designer program is used, hundreds of lines of code are generated and incorporated into one's program, labeled "do not touch", which does not aid the understanding of principles at all.

With the advent of Java and .NET, the possibilities for cross platform and cross language computing have increased greatly. However, libraries which support GUIs have, for the most part, not been included in this advance. For example, in the shared source version of .NET, the Windows.Forms API is specifically excluded [4]. There is therefore an opportunity to provide something that will travel better. Rather than design a new library – which would suffer from the same problems as other libraries – an alternative is to use XML for the specification. XML has the advantage that it is universal, and can be written by hand or emitted from a tool very easily.

The paper has two main sections. We start by defining a set of principles for GUIs. These are intended to be universal, and to be at a level where they could be taught in a lecture or two prior to the beginning of a section on GUI programming in an introductory course. We then follow with a section on the Views system, showing how the principles can be implemented, thus making GUIs reusable, adjustable and separately maintainable. We end by considering other work in the area to date.

2. GUI ELEMENTS
One of the problems of extracting a set of principles for GUIs is that many of the terms are already language specific. However, this confusion is not confined to GUIs: consider that we have case versus switch statements and base vs super classes. We will choose and stick to one set of terms, initially giving the alternatives. A GUI consists of the following five parts:

- controls (or components or widgets)
- a window (or form or frame)
- a menu bar
- layout
- interaction

A GUI is seen as a *window* on the screen and contains a number of different *controls*. Controls can, for example, be labels, buttons

SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.

or text boxes. The GUI very likely has a *menu bar* across the top which typically contains the name of the program or window and buttons for hiding, resizing and destroying the GUI display. Additional options can be added to the menu bar. The menu bar offers options similar to controls, but it is different in the way in which it is created and displayed.

The controls are said to have a certain *layout* which defines their position relative to each other and/or directly in the window. Both the choice of the controls and their correct layout are the concern of the programmer, even if a GUI design is specified.

Once the controls are on the screen, we *interact* with some of them via devices such as the mouse and keyboard. Moving the mouse over a control, and then clicking can make things happen (as with a button). At other times, we can be in a control that allows us to type from the keyboard. Other controls define areas for outputting data, including text, images, video and sound. It is this area of interaction that is the most challenging in GUI development, and to which we shall return in Section 3.

2.1 Controls

Most languages offer upwards of 30 or 40 controls, each with numerous options or attributes. These attributes can be size, color, font type and so on. The attributes are set at the time the control is created, but in most systems, they can be changed dynamically. For example, a text box created as

```
TextBox password = new TextBox();
```

can later have its background set to yellow by:

```
password.BackColor = Color.Yellow;
```

Enumerating all the controls and all their attributes, even for one language, is a large task, but in fact the beginning programmer should be introduced to the variety on offer, since the appropriate choice of a control is important. For example, if we want to have an image and be able to click on it, we could use a button and set the image attribute, or we could use a picture box, which can respond to double clicking on the image specified. The picture box has extra scaling facilities, so the developer would have to choose.

A set of principles regarding controls would contain a compendium indexed not only by control names, but by attributes, such as clickability. Then the programmer can see what controls have this attribute, and can make an appropriate choice. Not all languages will have the same set of matches for controls and attributes, but at least, having learnt the principles, the developer knows what to look for.

2.2 Layout

When developing a GUI, we can just add controls to the window as we think of them, but that would not usually make for a very pleasing arrangement. A key feature of GUI design is to group similar controls together. Thus a designer could stipulate that all the buttons be at the top, or at the bottom of the window. We can split the screen in half, and have input boxes on one side, and output on another, and so on. The question is, how does the developer manipulate controls like this? There are three options.

2.2.1 Drag and drop.

We can use a tool which allows us to create the code for a GUI by dragging a control from a list showing all the possibilities and dropping it onto the spot which looks right visually. We can change the placement of controls by dragging them with the mouse, and change their sizes and other attributes just as easily.

Such tools are actually very sophisticated pieces of software and take a lot of room on a computer. An example is Visual Studio which is built specifically for C# and other languages on the Windows platform. It generates C# code which uses advanced features such as events and delegates and is labeled “do not touch”. For a course aimed at teaching principles, such an approach is less than ideal.

2.2.2 Absolute positioning.

Without using drag and drop, we can write calls to library methods which precisely position a control down to the last pixel. The code to place a button at position x=300, y=150 would be something like the following:

```
Button submit = new Button();
submit.Location = new Point(300, 150);
```

which in itself is not difficult to write or understand, but working with many controls in absolute coordinates brings with it many extra small details to specify, and can become very long-winded. Another problem is that some screens these days are bigger than the earlier standard 600×800, in which case absolute positioning has to be done most carefully to obtain results which look good across a variety of computers.

2.2.3 Relative positioning

Relative positioning was popularized in the form of Java’s layout managers. They offer standard arrangements such as flow, border and grid, into which components can be placed, with the system handling the actual positioning of components with suitable gaps between them. For most GUIs they work very well, though they can have the same scaling problem mentioned above.

2.3 Interaction

Definition and layout of controls is mostly a static exercise. Interaction is dynamic, in that it will continue through the life of the program, and requires careful programming to cover all possibilities.

Creating an instance of window object causes the GUI to be displayed on the computer’s screen. The program can then wait for the user to do something, such as to click a button or type some text. The part of the program which receives such a prompt is known by the general term of an *event handler*. Event handlers are typically methods which are passed a parameter identifying the particular control that was activated. Then the handler can decide what to do about it. Active controls such as buttons should definitely have handlers, whereas potentially passive ones such as labels and images need not.

Event handlers are linked to controls early in the life of a program by means of the process of listening. Listeners essentially watch one or more controls and, when an action takes place, activate all registered event handlers. Once in a handler, we can interact with a control by calling methods defined to get data from it and put data into it. The selection of methods available depends on the language and library.

The interplay between listeners and handlers is the most complex part of GUI programming because it usually involves higher-order

programming constructs, such as delegates or callbacks, and the previously assumed sequential ordering of statements is disrupted. The confusion caused in the mind of the developer can be alleviated by a better division of responsibility, as discussed in the next section.

3. INDEPENDENT GUIs

Given that principles of GUIs can be explained, as in Section 2, the next challenge is to see whether these principles can be implemented in a language- and platform-independent way. The advantages of such a system would be:

- reuse of GUIs between systems,
- independent development of GUIs,
- better understanding of programming techniques,
- standardization of GUI notations.

The first three are in fact similar to advantages often quoted for objects. One of the biggest forces in the acceptance of object orientation in industry was its coupling to a standardized notation, and we look to developing the same for GUIs. It should be possible to have an assignment or a project which starts off: "given the following GUI design written in standard notation, write a GUI-based program/system to …". At present, the development of the GUI is definitely seen as an added extra, and very often developers are caught up in the conundrum of spending time on an aspect of the system which may not deliver much in the way of marks or revenue, but which is immensely satisfying and rounds off the project.

Once again we stress the difference between design and development. Programmers should be able to develop GUIs, but not all of them will be adept at good design. There should be a way of encapsulating GUI designs in specifications, in the same way as UML captures program design.

3.1 Views – a GUI system

We have developed a GUI notation and complete system to implement the principles described above. Views – a Vendor Independent Extensible Windowing System – consists of:

- an XML-based notation for specifying GUIs, and
- an engine for supplying listeners and interacting with event handlers.

Views is designed to be language and platform independent, although the choice of control names is closer to that of Microsoft Windows.Forms than to Java or other systems. The main engine runs in C# on Windows, but it has been ported to Java and to Tcl/TK, which makes it available for all the languages in the .NET platform, and on all the platforms on which .NET runs through the SSCLI (e.g. Unix, Macintosh). Other systems similar to Views are described in the section on Related Work.

Consider the development of a GUI as shown in Figure 1. We assume that a designer or builder program is used to drag and drop controls and set their attributes. The program will also usually permit the linking up of handler methods, or the programmer must do this himself or herself. At runtime, the controls are rendered (drawn) on the screen by the operating system, and interaction with them is passed on to the handlers. The developer is responsible for three parts of the process, and has to take on board hundreds of lines of generated code from the builder program.

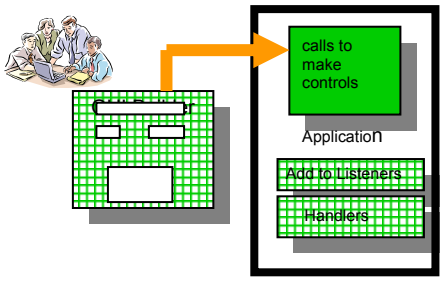


Figure 1. GUI development with a builder

In Views, we take a different approach, as shown in Figure 2.

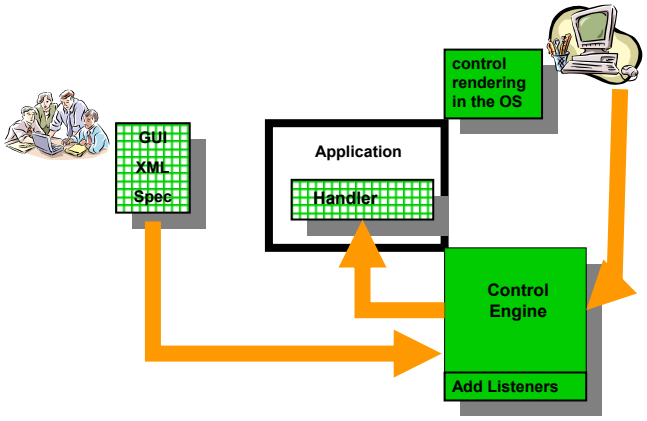


Figure 2. GUI development with Views

Here, the GUI is specified in XML. It is possible to do this fairly simply by hand, or one can have a tool which produces the XML. The effect for reuse and independence is the same. The XML is accepted, checked and used by the engine to cause the drawing of the controls in the window. The engine is also responsible for defining and maintaining the listeners for the controls specified. The developer's task is now reduced to just providing handlers, and there is no injected code in the application itself.

3.2 Example

Full details of the Views notation are given on the website [8] and in the book on programming which uses it [1]. The example that follows is very simple and is intended to be a vehicle for explaining the Views idea, rather than expounding on its considerable options and power.

We start with a simple program to read some numbers and calculate an exchange rate. In line-oriented mode in C#, this would be:

```
void TextGo() {
  Console.WriteLine("Currency Calculator");
  double euro, GBP;
  for (string c = "Y"; c!="N";
       c = Console.ReadLine()) {
    Console.Write("Paid on hols: ");
    euro = double.Parse(Console.ReadLine());
```

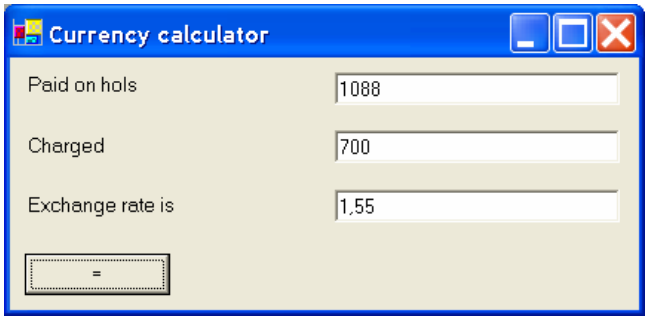
```
    Console.Write("Charged on credit card: ");
    GBP = double.Parse(Console.ReadLine());
    Console.WriteLine("Exchange rate is {0:G}",
        euro/GBP);
    Console.Write("More? ");
  }
}
```

We do not have room for a hand or builder-made version, but here is the Views version.

```
void GuiGo(string spec) {
  Views.Form f = new Views.Form(spec);
  double euro, GBP;
  for (string c = f.GetControl(); c!=null;
        c = f.GetControl()) {
    euro=double.Parse(f.GetText("eurobox"));
    GBP =double.Parse(f.GetText("GBPbox"));
    f.PutText("ratebox",
              (euro/GBP).ToString("f"));
  }
  f.CloseGUI();
}
```

When GuiGo is invoked with the following specification string as its parameter, the GUI shown below is displayed.

```
static string specEn =
  @"<form Text='Currency calculator'>
    <horizontal>
      <vertical>
        <Label text='Paid on hols'/>
        <Label text='Charged'/>
        <Label text='Exchange rate is'/>
        <Button Name=equals Text='='/>
      </vertical>
      <vertical>
        <TextBox Name=eurobox/>
        <TextBox Name=GBPbox/>
        <TextBox Name=ratebox/>
      </vertical>
     </horizontal>
</form>";
```

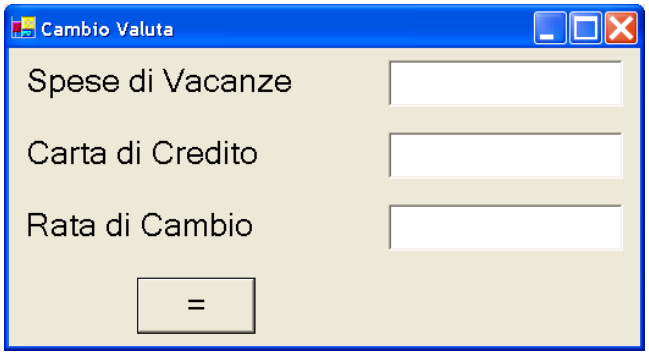


When GuiGo is called, the Views window (called a form) is first created as an object. Next there is a loop which gets the next control that has been activated and exits if the control is null. Views returns a null value if the close box on the window is clicked, thus uniting the behavior of the user with the program.

Thereafter we interact with the text boxes through GetText and PutText methods. The two programs are remarkably similar in their algorithm structure, but observe that there is no GUI data included in the second. That enhances *separation of concerns*, and means that both the design of the GUI and text in the window could change completely without changing the program.

Consider now the Views specification. Views works on relative position, but not via flow managers. Instead it has the concept of nested vertical and horizontal lists. Here we have a horizontal list of two items, and the first has four items and the second three. Each control is identified by an XML tag and followed by attributes. For dynamic controls, a name must be specified, which becomes the name by which the control is referred to in the program. All the other attributes are defaulted. The XML has been relaxed so that it is easier to write as a C# string constant and Views provides an extensive error checking process to ensure that GUI specifications can be debugged.

To show the power of an XML-based approach, we can supply a different specification string to the GuiGo method, and the following GUI will be displayed instead:



In this case, the font attributes were used to increase the font size, and the button was moved more centrally.

3.3 Event handling

This very simple example showed a GUI using sequential processing: Each item must be entered in turn, and is read and calculated as specified. GUI programming involves far more, as described in Section 2. Essentially, the user is in charge, and can activate controls in any order. The program is written in such a way as to be able to react to these controls seemingly at random. There are two ways in which this can be done:

- an event loop, and
- call back methods.

Views supports both, but advocates the first. Support for callbacks is supplied more for those who are already at ease with that technique. The event loop works by wrapping the statements in a switch statement, dividing them up as reflects the logic required. In our simple example, all of the action would fall under the equals control. To emphasize what an event loop is, we assume a second button, reset, has been added. The loop within GuiGo would become:

```
for (string c = f.GetControl(); c!=null;
     c = f.GetControl()) {
  switch (c) {
    case "reset":
      euro=1; GBP=1;
      f.PutText("eurobox",euro.ToString("f"));
      f.PutText("GBPbox",GBP.ToString("f"));
      break;
    case "equals":
      euro=double.Parse(f.GetText("eurobox"));
      GBP =double.Parse(f.GetText("GBPbox"));
      f.PutText("ratebox",
        (euro/GBP).ToString("f"));
```

```
            break;
          default: break;
        }
      }
```

The action is quite clear: the loop processes activated controls as they arrive, and the switch directs the action based on the name of the control. The name is the same name as that used in the Views specification. Thus, for example, we had

```
<Button Name=equals Text='='/>
```

In the C# code above, the use of switch on strings is particularly useful, but other ways of switching can also be employed in other languages. How Views provides for callbacks is discussed next.

3.4 Callbacks

As shown in Figure 2, Views handles the association between controls and handlers. Using the recommended programming style in [1], as shown in the example, there is one point of contact between the engine which listens for events and the program. That point is the for statement, which has the clause

```
string c = f.GetControl();
```

In the alternative callback model, which is that used by Java and C# (when not supported by Views), there are potentially many points of contact. Each control must be linked to a method on creation. In the builder for C#, a statement for a simple button would be generated as follows:

```
this.button1.Click += new
   System.EventHandler(this.button1_Click);
```

Unpacking this statement, we have the control button1, which has a listener called Click. The listener is represented in C# by a special object called an *event*. The event handler method which the programmer must supply is called button1_Click. The method is linked up via the delegate EventHandler which accepts registrations of methods for events. The signature of button1_Click is also prescribed as:

```
private void button1_Click(object sender,
   System.EventArgs e)
```

This syntax is different to Java's which has system wide event handlers for certain types of events, and within each handler, the programmer must sort out what control caused the event.

If one wished to use this model in Views, the process is simple. Each attribute available to a control can be set in the specification. Thus it is possible to augment the button declaration as:

```
<Button Name=equals Text='='
             Click=equalsHandler/>
```

However, there are potentially race conditions within the engine if both models are used simultaneously. We are still investigating whether these can be satisfactorily resolved, or even whether it makes sense to have both models operational at the same time.

4. RELATED WORK

There has been little or no work done on the categorization of GUI controls and the establishment of principles of layout handling. In other areas, APIs had more attention paid to them, as in the case of the database library which has been incorporated into a new language, XEN [2].

Interaction has been studied in more depth and the most thorough work is that of Meyer [3] who proposes a simple to use publish-subscribe library and compares it to the C# delegate model discussed here.

There are two other major efforts for introducing independent XML based GUI specifications to languages. The first is XUL [7] which is Java-oriented. It has controls which reflect those of Java, and the handlers are written in the specification itself, in JavaScript. Its platform independence is thus achieved via Java. The other is UIML [5] which also has a Java flavor, and whose handlers are intended to be written in XML, which is almost impossible to do without tools. The difference between these two initiatives and Views is where the line is drawn for the specification of handlers. In Views, we regard them as being part of the application, solely within the realm of the programmer. The specification and layout of the GUI is what is XML-based, and could be written separately and reused.

5. CONCLUSIONS

We have proposed that GUIs should be viewed as having principles related to their elements (controls, menu bars, positioning) and their interaction with the user. While principles regarding good GUI design are well-established, principles regarding which programming structures to use with them are not. Moreover, some of these, particularly for interaction, are too complex for the average programmer, and the approach of using a GUI builder hides the very principles that one wishes to expose.

Our Views system espouses GUI principles and provides a clear separation of concern between the display of the GUI and the interaction with the program. It is therefore possible to separately develop GUIs, and to reuse, adapt and maintain them easily.

Our work on the initial Views system has been completed. It included some intricate programming with reflection, XML parsing and regular expressions. We are now concentrating on unifying the availability of Views on the SSCLI platform of .NET and of re-investigating our earlier Java version.

ACKNOWLEDGMENTS

We thank Microsoft Research for their interest in and generous support, our students who assisted with programming and testing the system, and the NFR for financial support.

REFERENCES

[1] Bishop, J., and Horspool, N**. C# Concisely**, Addison Wesley, 2004.

[2] Meijer, E., and Schulte. W. *Xen and the Art of Coherence Maintenance* (in preparation).

[3] Meyer, B., *The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design*, to appear in **Festschrift in Honor of Ole-Johan Dahl**, eds. Olaf Owe et al., Springer-Verlag, LNCS 2635, 2003.

[4] Stutz, D., Neward, T., and Shilling, G. **Shared Source CLI Essentials**, O'Reilly, 2003.

[5] UIML website. http://www.uiml.org/index.php

[6] Views website. htpp://www.cs.up.ac.za/rotor

[7] XUL website. http://www.xulplanet.com/

[8] C# Concisely website. http://csharp.cs.uvic.ca