Object-Oriented Graphical User Interface

Entities of graphical user interface (GUI), such as buttons and combo-boxes, are generally presented as a set of classes and their sub-classes. A user program includes such definitions and needs to link with libraries provided with a particular implementation of GUI. Some libraries support multiple platforms by acting as a pass-through to several platform-specific libraries. Nonetheless, the user code is aware of the library and includes calls to it.

For responding to user input the general pattern is to define a special global function and register it with the platform. This function is usually called input wait-loop. One then accumulates all the handling of GUI input within this global function.

The next progress was made when tools for creating GUI were introduced. Nevertheless, these tools generate the code that a user program must be aware of, and make references to it in various parts of the program. In particular, the C defines for referencing the GUI entities is quite visible in such programs.

Although in the approach just described GUI entities are treated as objects, the overall implementation is not object-oriented. Furthermore, the user program is aware of the code of the library that provides the implementation for the GUI elements.

In this article we introduce the object-oriented approach of Z++ for dealing with GUI (OOGUI). In Z++ the code implementing GUI entities is not visible in user program, and there is no global input wait-loop.

Canvas

The construction of GUI in Z++ Visual is done entirely with its GUI-Maker. The area on which GUI is designed is called a canvas, which is a representation of a run-time dialog. This is similar to other tools used in other languages. However, Z++ GUI-Maker only generates a simple structure for inclusion in user code, which is also called a canvas. Below is an example of a generated canvas, with two buttons and a field for entering text.

```
canvas Sample
  button Done;
  button Clear;
  field Text;
end;
```

Canvas is a Z++ type that is only generated by its GUI-Maker. The compiler recognizes the terms button and field only within the scope of a canvas. Otherwise these terms are not reserved.

The name of the above canvas is Sample. The identifiers Done, Clear and Text are names of those GUI entities and may appear on them at run-time. For instance the button named Done will be labeled as Done. On the other hand a field will have no label. The user code

will include the above definition of generated canvas, and as we will illustrate, will reference the canvas Sample and the identifiers of the entities in it.

Frame

A Z++ frame is identical to a class with a few differences. Basically, a frame must be associated to a canvas, and must have a special method called its instinct method. The instinct method is special in the same sense as a constructor or destructor.

The identifier for an instinct method is the same as the name of the frame, preceded with the $ sign. This is similar to the identifier for a destructor except the symbol ~ is replaced with $. The signature of an instinct method is fixed and will be discussed later.

A frame is defined by the engineer, and is associated with a canvas. Thus, the same canvas may be associated with different frames for different uses. Below is an example of a frame associated with the canvas Sample. The association operator is a colon followed with the equality sign.

```
frame My_frame := Sample
  // members, methods
public:
  // constructors, destructor
  // instinct method
end;
```

There are several advantages to this approach. Clearly, the handling of a dialog is localized because its canvas is associated to a frame. The instinct method of the frame serves the same purpose as the input wait-loop.

A frame can have other frames as its members. This simplifies managing parent-child relationship. A member frame is just the child of its owning frame. Since a frame can exchange data with its members, parent and its children can communicate without the use of global data.

The interface include file

Before illustrating the use of instinct method we need to take a look at the Z++ system header file, interface.h. This file contains the definitions of types that are used in an instinct method. Below is a sample of types of events.

```
enum interfaceEventSignals {
      _IES_Draw_Signal,
      _IES_Erase_Signal,
      _IES_Pen_Tap_Signal,
      _IES_Pen_Hold_Signal,
      _IES_Pen_Drag_Signal,
      _IES_Pen_Drop_Signal,
      _IES_Invalid_Signal
};
```

Events and signals in Z++ are of enumeration type so one can extend them to include user-defined events and signals. Z++ allows extending enumerations, which is covered in other white papers.

The signature of an instinct method is a reference to the following type, defined in system header file interface.h.

```
struct interfaceEventType
      ushort x;
      ushort y;
      ushort entity
      interfaceEventSignals Event
end;
```

The x and y coordinates of a mouse-click are rarely used in user code. The member entity is a canvas entity like button or field, as we will illustrate shortly.

The instinct method

Below is a skeletal example for an instinct method of the frame My_frame.

```
My_frame::$My_frame(interfaceEventType& e)

  switch(e.Event)

    case _IES_Draw_Signal:
    case _IES_Erase_Signal:
    case _IES_Pen_Tap_Signal:

      select(e.entity) //select statement starts here

        case Done:
        case Clear:
        case Text:

      endselect; //select ends here

  endswitch;
end;
```

Obviously, there can be statements before and after the switch statement. However, this simple code illustrates the pattern for the body of an instinct method. First, one switches over the types of events. Second, for each event of interest (for instance pen-tap in our example), one uses a select statement over the entities of interest in the associated canvas.

The select statements can only appear in the body of an instinct method. Its pattern is similar to the switch statement. However, the case labels for a select statement are identifiers of the entities in a canvas.

Entering GUI mode

When an instance of a frame is created (i.e. when its declaration is elaborated), nothing is drawn. The canvas associated with a frame is drawn by applying the draw operator $ to the instance of the frame, as shown below.

```
My_frame MF; // create an instance of frame
$MF;         // draw the canvas of the frame and enter GUI mode
// this is reached after the canvas is erased
```

Once the canvas is drawn, control does not move to the next statement until the canvas is erased. This is the GUI mode, the state in which all user input is directed to the instinct method of the frame. GUI mode ends when the canvas is erased, and only then the control moves to the statement following the draw statement, as illustrated above.

This approach makes drawing various canvases based on conditions quite intuitive. Furthermore, one can simply draw multiple canvases in multiple threads. Since each canvas is controlled via the instinct method of its associated frame, the code is localized, decoupled and intuitive.

Illustrating instinct method

Let us illustrate the events shown in the body of the instinct method of the frame My_frame. When the canvas of a frame is drawn the frame receives the draw event. This can be ignored, or used for initialization such as populating lists or writing text in fields.

Observe that entities in canvas are reached via the scope operator, as usual. For instance the field Text is reached as Sample::Text. The ~ operator applied to a field erases its contents. Operator << simply writes the text to the field, as expected.

Erase takes two steps. First, when the button Done is tapped the erase signal is sent to the canvas. This results in the erase event being sent to the frame. The frame erases itself by applying the operator $ to itself, i.e. $self. At this point the frame is erased and GUI mode for this frame ends. As discussed earlier, control of the thread moves to the next statement following the one that drew the frame, that is $MF in our example shown below.

```
My_frame::$My_frame(interfaceEventType& e)

  switch(e.Event)

    case _IES_Draw_Signal: // initialization
        ~Sample::Text; // first clear the field
        Sample::Text << “Hello World!”;  // then write to it

    case _IES_Erase_Signal:
        $self;  // erase canvas of this frame and exit GUI mode
        return;

    case _IES_Pen_Tap_Signal:

      select(e.entity) //select statement starts here

        case Done:
            Sample <- _IES_Erase_Signal; // tell canvas to erase itself

        case Clear:
            ~Sample::Text; // clear the field

        case Text:
            // nothing to do here

      endselect; //select ends here

  endswitch;
end;
```

Conclusion

The presence of GUI related code in a program provides the opportunity for all kinds of versatile third-party libraries. However, much of the most useful features can be automated through the use of visual tools.

The benefits of a GUI-Maker become greater with the omission of GUI code from a program. The lack of reference to such code in a program reduces the development and testing times. Furthermore, a GUI-Maker can generate optimal code for multiple platforms without having to touch the program written by engineers.

The object-oriented approach illustrated in this note facilitates localization, decoupling and cohesion. A frame can be focused on a certain aspect of a program, and when modifying the code of a frame, other parts of the program remain unaffected. Furthermore, parent-child relationship and data exchange among them becomes more intuitive as opposed to the use of global data.