Modelling Systems With Components

by Paul Warren

When I moved from Borland Pascal 7 to Delphi I, like probably every other Pascal programmer, was suitably impressed with the IDE, the RAD concept and the cool components. I was, however, mildly disappointed that the elegance of OOP seemed slightly removed from the developer. Of course this impression was patently untrue as I discovered during a recent project. If anything, Delphi components are more cleverly encapsulated objects than anything I've previously seen.

Decision Trees

I needed to create a number of decision trees for my project in order to categorise materials according to their toxicity, flammability and corrosivity. My first decision tree went something like Listing 1.

```
if variable > 38 then
  class := ’B1’;
if variable > 48 then
  class := ’B2’
else
  class := ’No classification’;
```

➤ Listing 1

A short while later I found myself writing virtually the same lines of code for the next decision tree. At this point I decided to create a generic decision tree object.

The approach I took was to create a component which I could program at design time to filter any input to the appropriate category. Since all decisions would be simply *Yes/No* I started by calling my component a `TBooleanNode`. I gave it an `Enabled` field (to disable nodes), an `FInput` field (to hold the input), a `Criteria` field (to hold the comparison value), and an `Operator` field (in case I wanted different comparisons). Listing 2 shows the original code.

```
TOperator =
  (opEquals, opGreaterThan, opGreaterOrEqual, opLessThan, opLessOrEqual);
TBooleanNode = class(TComponent)
  private
    FInput: Single;
    FEnabled: Boolean;
    FCriteria: Single;
    FOperator: TOperator;
  public
    constructor Create(AOwner: TComponent); override;
    procedure Run;
  published
    property Input: Single read FInput write FInput;
    property Enabled: Boolean read FEnabled write FEnabled default True;
    property Criteria: Single read FCriteria write FCriteria;
    property Operator: TOperator read FOperator write FOperator;
  end;
constructor TBooleanNode.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEnabled := true;
end;
procedure TBooleanNode.Run;
begin
end;
```

➤ Listing 2

I quickly realised all I would get out of this component would be a Yes/No answer for each input. Sure, I could reset `FCriteria` programmatically at run time and call `Run` again for a new Yes/No answer, but that would be messy and a lot of code for a such a simple task.

Another solution might be to create an `FNodes` field to hold the number of decisions and make `FCriteria` an `array[1..FNodes]` holding the criteria for the decisions. Unfortunately I would then need an `array[1..FNodes]` of operators or assume all comparisons would use the same operator. Not much hope here.

Somewhere in the development process I began to wonder if my `TBooleanNode` could communicate to another instance of itself. Then, I could connect any number of `TBooleanNode` instances together and each would evaluate the input and send the result to the next node. I added an `FYesPipe` and `FNoPipe` of type `TBooleanNode` to the component and modified the `Run` method to evaluate the input and pass it along to the next component in the chain. Listing 3 shows the code I used at this stage.

If you look at the `Run` method closely you will realise there has to be another `TBooleanNode` connected or you get a GPF when `YesPipe.Input` is called. You could test `YesPipe` to see if it's not `nil` but I thought a `TEndNode` component which could be connected to a `TBooleanNode` might be a better approach, especially since the `TEndNode` component could take care of reporting the result of the decision tree.

To complete the project I needed to create a base class for both the `TBooleanNode` and the `TEndNode` so they could be used interchangeably. I created a `TNode` object as the base class and moved the `FInput`, `FYesPipe` and `FNoPipe` fields to `TNode`. I created a procedure `Run; virtual; abstract;` for `TNode` and re-declared `TBooleanNode` and the new `TEndNode` as `class(TNode)`. Listing 4 shows the working code for the components.

In order to report the results of running these connected components I added an `FResultStr` field and an `AfterRun` event to `TEndNode`. When these components are compiled into the component library they can be connected into nearly any kind of decision tree you may want. You just place `TBooleanNodes` on your form and set the criteria and operators for each node and join them as needed. End each possible path with a `TEndNode` holding the appropriate string and connect all the `TEndNode AfterRun` events to the same method. Listing 5 shows an example of the code to display the `ResultStr` in a label. There are two demo programs included on the free disk which show the versatility of these components. Figure 1 shows a screen capture of the design time form for the `BooleanProj` demo with the nodes added and Figure 2 shows the demo application displaying the result of running the decision tree.

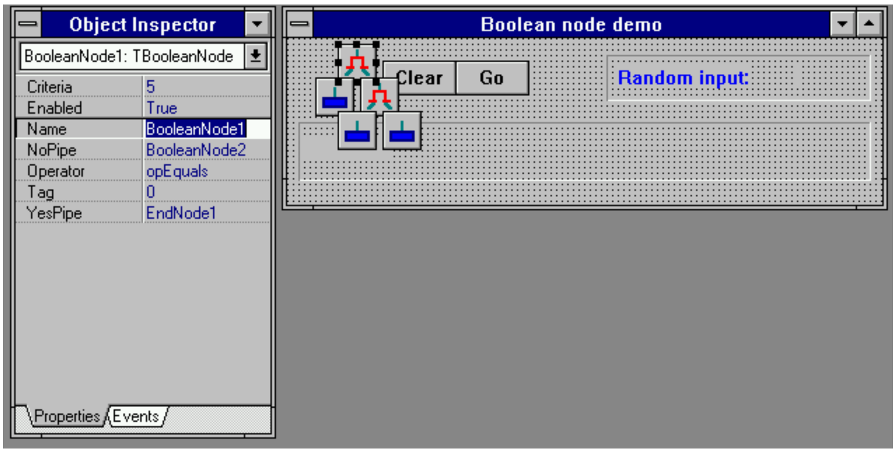
Real Properties

Only one problem remained in implementing this system. Occasionally I would have to use floating point types in the decision tree. This would be easy if all the machines I have to support had 80x87 co-processors. The `FCriteria` and `FInput` properties could simply be re-declared as type `Double`.

Unfortunately there is no built-in support for property editors of type `real` (I have had this problem with other components as well). The solution is to write a property editor which emulates `real` types.

Listing 6 shows the code for a simple property editor which acts as an editor for `real` types by accepting a string input and checking to see if it can be converted to a `real`. This allows the user to edit the property in the Object Inspector as if it was a `real`. With the addition of a `protected`

➤ *Figure 1*



BooleanNode1: TBooleanNode

Criteria	5
Enabled	True
Name	BooleanNode1
NoPipe	BooleanNode2
Operator	opEquals
Tag	0
YesPipe	EndNode1

➤ *Listing 3*

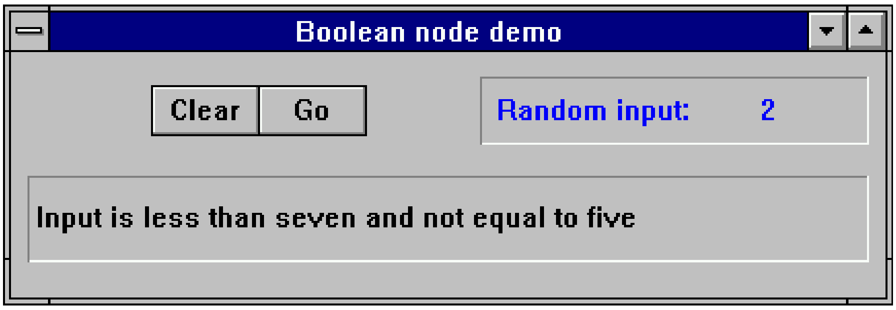
```
TBooleanNode = class(TNode)
  private
    FInput: Single;
    FEnabled: Boolean;
    FCriteria: Single;
    FOperator: TOperator;
    FYesPipe: TBooleanNode;
    FNoPipe: TBooleanNode;
  protected  { Protected declarations }
  public
    constructor Create(AOwner: TComponent); override;
    procedure Run;
  published
    property Input: Single read FInput write FInput;
    property Enabled: Boolean
      read FEnabled write FEnabled default True;
    property Criteria: Single
      read FCriteria write FCriteria;
    property Operator: TOperator
      read FOperator write FOperator;
    property YesPipe: TNode read FYesPipe write FYesPipe;
    property NoPipe: TNode read FNoPipe write FNoPipe;
  end;
constructor TBooleanNode.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEnabled := true;
  FYesPipe := nil;
  FNoPipe := nil;
end;
procedure TBooleanNode.Run;
begin
  case operator of
    opEquals :
      if FInput = FCriteria then begin
        YesPipe.Input := FInput;
        YesPipe.Run;
      end else begin
        NoPipe.Input := FInput;
        NoPipe.Run;
      end;
    opGreaterThan :
      if Input > Criteria then begin
        YesPipe.Input := Input;
        YesPipe.Run;
      end else begin
        NoPipe.Input := Input;
        NoPipe.Run;
      end;
    opGreaterOrEqual :
      if Input >= Criteria then begin
        YesPipe.Input := Input;
        YesPipe.Run;
      end else begin
        NoPipe.Input := Input;
        NoPipe.Run;
      end;
    opLessThan :
      if Input < Criteria then begin
        YesPipe.Input := Input;
        YesPipe.Run;
      end else begin
        NoPipe.Input := Input;
        NoPipe.Run;
      end;
    opLessOrEqual :
      if Input < Criteria then begin
        YesPipe.Input := Input;
        YesPipe.Run;
      end else begin
        NoPipe.Input := Input;
        NoPipe.Run;
      end;
  end; {case}
end;
```

property `AsReal`, for run-time use, we can use `real` types in any component. Any time a property is declared as type `TRealStr` it will work in the Object Inspector as if it were a `real`.

The final step then was to change the `FInput` property to a `private` field `Input: real;` and add an `InputAsReal` property in `TNode`. Then I changed the `FCriteria` property to a `TRealStr` and added a `CritAsReal` property to `TBoolean` node. You will find the complete code on the disk with this issue. As you will see from the demos you can set the `FCriteria` property to any valid floating point number within the range for type `real` even without a co-processor. (As a bonus you will find a useful editor for `real` types on the disk).

I was pleased with the results of this little project because the code was reusable and generic, the nodes could be programmed at design or run time and the results of each decision tree run could be output in different ways. I use a monolog speech component to output the result verbally in one implementation. On calling `Run` for the top node the input automatically cascades down the decision tree in a way which seems almost human.

➤ *Figure 2*



Input is less than seven and not equal to five

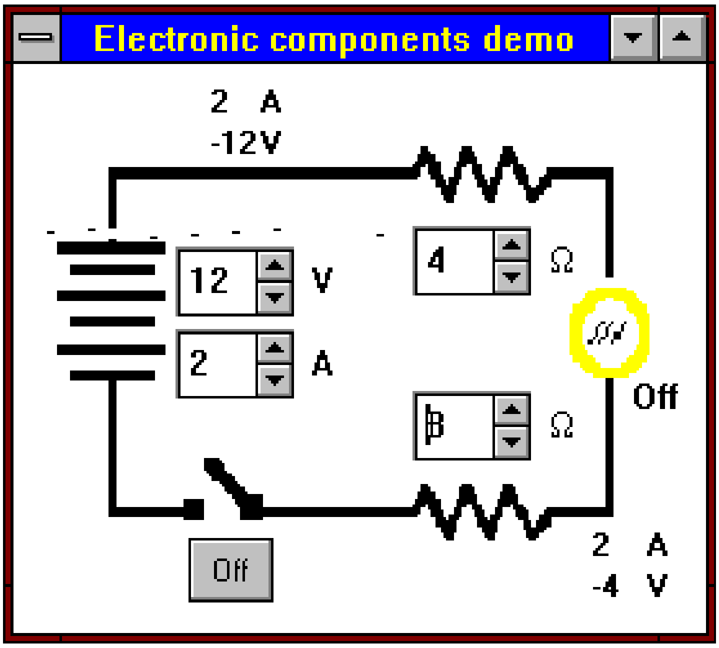
➤ *Listing 5*

```
procedure TForm1.EndNodeXAfterRun(Sender: TObject);
begin
  Label1.Caption := (Sender as TEndNode).ResultStr;
end;
```

➤ *Listing 4*

```
type
  TNode = class(TComponent)
  private
    FInput: Single;
    FYesPipe: TNode;
    FNoPipe: TNode;
    procedure Run; virtual; abstract;
  published
    property Input: Single read FInput write FInput;
  end;
  TOperator = (opEquals, opGreaterThan, opGreaterOrEqual,
               opLessThan, opLessOrEqual);
  TBooleanNode = class(TNode)
  private
    FEnabled: Boolean;
    FCriteria: Single;
    FOperator: TOperator;
  protected { Protected declarations }
  public
    constructor Create(AOwner: TComponent); override;
    procedure Run; override;
  published
    property Enabled: Boolean
      read FEnabled write FEnabled default True;
    property Criteria: Single read FCriteria write FCriteria;
    property Operator: TOperator
      read FOperator write FOperator;
    property YesPipe: TNode read FYesPipe write FYesPipe;
    property NoPipe: TNode read FNoPipe write FNoPipe;
  end;
  TEndNode = class(TNode)
  private
    FResultStr: string;
    FAfterRun: TNotifyEvent;
    procedure SetAfterRun(Value: TNotifyEvent);
  protected
    procedure After; dynamic;
  public
    procedure Run; override;
  published
    property ResultStr: string
      read FResultStr write FResultStr;
    property AfterRun: TNotifyEvent
      read FAfterRun write SetAfterRun;
  end;

constructor TBooleanNode.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEnabled := true;
  FYesPipe := nil;
  FNoPipe := nil;
end;

procedure TBooleanNode.Run;
begin
  case operator of
    opEquals :
      if Input = FCriteria then begin
        YesPipe.Input := FInput;
        YesPipe.Run;
      end else begin
        NoPipe.Input := FInput;
        NoPipe.Run;
      end;
    opGreaterThan :
      if Input > FCriteria then begin
        YesPipe.Input := FInput;
        YesPipe.Run;
      end else begin
        NoPipe.Input := FInput;
        NoPipe.Run;
      end;
    opGreaterOrEqual :
      if Input >= FCriteria then begin
        YesPipe.Input := FInput;
        YesPipe.Run;
      end else begin
        NoPipe.Input := FInput;
        NoPipe.Run;
      end;
    opLessThan :
      if Input < FCriteria then begin
        YesPipe.Input := FInput;
        YesPipe.Run;
      end else begin
        NoPipe.Input := FInput;
        NoPipe.Run;
      end;
    opLessOrEqual :
      if Input < FCriteria then begin
        YesPipe.Input := FInput;
        YesPipe.Run;
      end else begin
        NoPipe.Input := FInput;
        NoPipe.Run;
      end;
  end;
end;

procedure TEndNode.SetAfterRun(Value: TNotifyEvent);
begin
  FAfterRun := Value;
end;

procedure TEndNode.After;
begin
  if Assigned(FAfterRun) then FAfterRun(Self);
end;

procedure TEndNode.Run;
begin
  After;
end;
```

➤ *Figure 3*



```
type
  TRealStr = string[20];
  TRealStrProperty = class(TStringProperty)
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
function TRealStrProperty.GetValue: string;
begin
  Result := LRTrim(Chop(GetStrValue, 20));
end;
procedure TRealStrProperty.SetValue(const Value: string);
var
  Temp: string[20];
  code: integer;
begin
  Temp := LRTrim(Value);
  StrToReal(code, Value);
  if code <> 0 then
    MessageDlg('Value must be a real number', mtError, [mbOk], 0)
  else
    SetStrValue(Value);
end;
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TRealStr), nil, '', TRealStrProperty);
end;
```

➤ *Listing 6*

What Next?

This started me thinking in two different directions: modelling other systems, such as neural nets and fuzzy logic, and providing components to my users in the same way Delphi provides components to developers.

I haven't proceeded with the former yet, but I have started experimenting with the latter. Also included on the disk is a suite of components which mimic electronic components. There is a `TConductor`, a `TBattery`, `TSwitch`, `TResistor` and `TLamp`. They are implemented in much the same way as `TBooleanNode` and `TEndNode`. Figure 3 shows the example.

I fully realise the behaviour of these components is not yet correct, but they do illustrate well the idea of modelling real life systems with components. I hope to return to this theme in another article, so watch this space!

The real challenge of course is to develop a system of providing modelling components to the end user. This requires giving these components a visual representation, creating them at run time and providing a property interface to the user. The results of this effort will be quite astounding. I can easily envision an electronics lab for students where complete circuits can be created at runtime to test theories and predict behaviour.

The more I use Delphi the greater the potential I can see using this superb language and IDE. I have recently wondered if Delphi is a true case of the whole being more than the sum of its parts. Does Borland really know what they have created?

The only complaint I have now about Delphi is that it has caused me to explore in too many directions at the same time and I don't have enough time to explore them all. Please help! If this article starts anyone toward creating end-user components please publish your code and let me know how you accomplished it. Oh, and if anyone creates an artificial intelligence do make sure it's benign...!

Paul Warren runs HomeGrown Software Development in Langley, British Columbia, Canada and can be contacted by email at hg_soft@haven.uniserve.com

Installing The Components

To install the components which I've discusssed in this article you will need to copy the following files from the disk into a directory on Delphi's search path:

NODES.PAS
NODES.DCR
ENHEDITS.PAS
ENHEDITS.DCR
ELECTRIC.PAS
ELECTRIC.DCR
STRLIB.PAS

then using Install Components (first carefully backing up your COMPLIB.DCL...!) install the files:

NODES.PAS
ENHEDITS.PAS
ELECTRIC.PAS

When your COMPLIB.DCL has recompiled you can then try out the example projects included on the disk.