Under Construction: TRuleBase Component

by Bob Swart

Last month, we built a first version of a limited inference engine, based on `TFact` and `TRule` classes and the forward and backward chaining algorithms. This time, we're going to expand these to true `TFactBase` and `TRuleBase` components, including some design time supporting property editors and component editors.

Facts

We designed the `TFact` object to work with facts that could have three values: Yes, No or Unknown. This is of course a limitation when reasoning, since we often need information which is expressed in other ways, such as the date of birth or gender of a person. These are important knowledge issues when it comes to life insurance, for example, where generally females live longer than males.

Therefore, the definition of `TValue` would need to change from a finite set of possible answers to, for example, a simple `ShortString`. Anything, including numerical information (such as income and debts, when it comes to credit assessment) can be stored in strings. Note that even Delphi's own database components seem to feel that way, since `TFields` almost all have an `AsString` property (except for big fields, such as BLOBs or Memos). This yields the modified TFact class shown in Listing 1.

```
Type
{$IFNDEF WIN32}
  ShortString = String;
{$ENDIF}
  TValue = ShortString;
  TFact = class(TObject)
  private
    FFact: Integer;
    FGoal: Boolean;
    FName: TName32;
    FValue: TValue;
    FQuestion: ShortString;
  protected
    constructor Create(Table: TTable); virtual;
  public
    property Fact: Integer read FFact;
    property Goal: Boolean read FGoal;
    property Name: TName32 read FName;
    property Value: TValue read FValue write FValue;
    property Question: ShortString read FQuestion;
  end {TFact};
```

➤ *Listing 1*

TFactBase

One fact is seldom enough when reasoning. We often need an entire database of facts, called a *factbase*. While it would be unwise to have a component for every fact in the database, it might be helpful to have a component that encapsulates the entire factbase: the `TFactBase` component.

First of all, we need to consider the ever recurring question of inheritance versus delegation. The facts are stored in a table, so we need a `TTable` component. But do we derive from this component in order to create our `TFactBase` component (inheritance), or do we simply use a `TTable` (as a field or property) in our new component `TFactBase` (delegation)? In this case, as in many others in fact, I prefer the delegation model over the inheritance model. Besides, when just using a `TTable`, we won't need to try to hide its properties in the Object Inspector from the (design time) user. See Listing 2.

```
Type
  TFactBase = class(TComponent)
  private
    FActive: Boolean;
    FFactBase: TFileName;
    FNumFact: Integer;
  protected
    FactTable: TTable;
    Facts: Array[0..MaxFact] of TFact;
  protected
    procedure SetFactBase(NewFactBase: TFileName);
    procedure SetActive(NewActive: Boolean);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  public
    procedure Open; virtual;
    procedure Close; virtual;
  public
    procedure NewFactBase;
    procedure Reset;
  published
    property Active: Boolean read FActive write SetActive;
    property FactBase: TFileName read FFactBase write SetFactBase;
    property NumFact: Integer read FNumFact;
  end {TFactBase};
```

➤ *Listing 2*

Since we're using a delegation model, the constructor must create the `FactTable` field of type `TTable`. We cannot open the table, of course, since we don't know the `DatabaseName` and `TableName` properties. These are derived from the `FactBase` filename property.

The destructor is used to close the `FactTable` (if it was open), which also cleans up the `Fact` classes that were used. Then the `FactTable` itself is freed, followed by the inherited `Destroy`. This way, we're sure not to leak any memory (and we can always use MemMonD or Memory Sleuth to check, remember?). See Listing 3.

The FactBase property is a fully qualified filename, which must be dissected into a path (DatabaseName) and filename (TableName) for the hidden FactTable field. This is done in the method SetFactBase, which first needs to close the dataset. And since we don’t want to close the dataset unnecessarily, we check to see whether or not the NewFactBase is different to FFactBase. See Listing 4.

Much like a regular TTable, we can set the Active property of TFactBase to True, which means that Facts are allocated and read from the FactTable into memory and we’re ready to do something with them (like reasoning, which comes next). We can also set the Active property to False, deactivating the FactBase, which means closing the FactTable and freeing the Fact classes that were allocated earlier.

Note that this is basically the code which we used in the initialization section of the Facts unit last month. Only now we’ve really encapsulated it into a component, so we can use multiple factbases.

There is one special case which we must take care of: loading a TFactBase from a stream file where the Active property is set to True. Since the properties are read in alphabetical order, the Active property is read before the FactBase property (containing the DatabaseName and TableName property values combined), which results in an exception when trying to open the FactTable without a valid DatabaseName and TableName (they are still blank). There are two ways to avoid the exception. The first would be to give the FactBase property a name which comes before the Active property. The other possible solution is simply to ignore the Active property when reading the component and leave it set to False. We can do that by looking at ComponentState and checking if csReading is in this set. If so, then we do nothing and so leave the Active property False. This is not exactly how a TTable works, but it’s enough functionality for now (we can always set the Active property to True in the OnCreate event of our form). See Listing 5.

```
constructor TFactBase.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FactTable := TTable.Create(Self)
end {Create};
destructor TFactBase.Destroy;
begin
  Close;
  FactTable.Free;
  FactTable := nil;
  inherited Destroy
end {Destroy};
```

➤ Listing 3

```
procedure TFactBase.SetFactBase(NewFactBase: TFileName);
begin
  if NewFactBase <> FFactBase then begin
    Close;
    FactTable.DataBaseName := ExtractFilePath(NewFactBase);
    FactTable.TableName := ExtractFileName(NewFactBase);
    FFactBase := NewFactBase
  end
end {SetFactBase};
```

➤ Listing 4

```
procedure TFactBase.SetActive(NewActive: Boolean);
var
  i: Integer;
begin
  if not (csReading in ComponentState) then { skip loading }
  if NewActive <> FActive then begin
    if NewActive then begin
      FactTable.Open;
      FactTable.First;
      while not FactTable.Eof do begin
        if FactTable.FieldByName(’Fact’).AsInteger <> FNumFact then
          raise Exception.Create(’Error: facts are not sorted...’);
        Facts[FNumFact] := TFact.Create(FactTable);
        FactTable.Next;
        Inc(FNumFact)
      end;
      FActive := True
    end else begin
      { Close }
      FactTable.Close;
      for i:=0 to Pred(FNumFact) do begin
        Facts[i].Free;
        Facts[i] := nil
      end;
      FNumFact := 0;
      FActive := False
    end
  end
end {SetActive};
```

➤ Listing 5

```
procedure TFactBase.NewFactBase;
begin
  with FactTable do begin
    Active := False;
    TableType := ttParadox;
    TableName := FFactBase;
    with FieldDefs do begin
      Clear;
      Add(’Fact’, ftInteger, 0, TRUE);
      Add(’Goal’, ftBoolean, 0, TRUE);
      Add(’Name’, ftString, 32, TRUE);
      Add(’Question’, ftString, 255, FALSE)
    end;
    with IndexDefs do begin
      Clear;
      Add(’index’, ’Fact’, [ixPrimary,ixUnique])
    end;
    CreateTable
  end
end {CreateFACTS};
```

➤ Listing 6

Again, like the TTable component, we can assign a value to the Active property of our TFactBase

```
Type
  TRule = class(TObject)
  private
    FRule: Integer;
    FCF:  SmallInt;
    FFact: Integer;
    FValue: TValue;
    FComments: ShortString;
  protected
    FFired: Boolean;
    constructor Create(Table: TTable); virtual;
  public
    property Rule: Integer read FRule;
    property CF:  SmallInt read FCF;
    property Fact: Integer read FFact;
    property Value: TValue read FValue;
    property Fired: Boolean read FFired write FFired;
    property Comments: ShortString read FComments;
  end {TRule};
```

➤ *Listing 7*

```
Type
  TRuleBase = class(TComponent)
  private
    FActive: Boolean;
    FRuleBase: TFileName;
    FFactBase: TFactBase;
    FNumRule: Integer;
  protected
    RuleMax: Integer;
    RuleTable: TTable;
    Rules: Array[0..MaxRule] of TRule;
  protected
    procedure SetFactBase(NewFactBase: TFactBase);
    procedure SetRuleBase(NewRuleBase: TFileName);
    procedure SetActive(NewActive: Boolean);
  protected
    function TestRule(RuleNr: Integer): Boolean;
    procedure FireRule(RuleNr: Integer);
    function Conclude(RuleNr, FactNr: Integer): Boolean;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  public
    procedure Open; virtual;
    procedure Close; virtual;
  public
    procedure NewRuleBase;
    procedure Reset;
  public
    function Forwards: Integer;
    procedure Backwards(Goal: Integer);
  published
    property Active: Boolean read FActive write SetActive;
    property NumRule: Integer read FNumRule;
    property RuleBase: TFileName read FRuleBase write SetRuleBase;
    property FactBase: TFactBase read FFactBase write SetFactBase;
  end {TRuleBase};
```

➤ *Listing 8*

component, or use the Open and Close methods. which merely set the value of Active. Since both are assigning values to the component's property (not the private FActive field), we know for certain that the SetActive method is called.

Suppose there is no initial factbase to work with. Trying to open a factbase would raise an exception. We need to make sure we can create a new factbase when needed, which is why we need the method NewFactBase (Listing 6).

Finally, in order to perform more than one session with the same factbase without having to re-read it into memory, we need a procedure to reset all fact values to Unknown: TFactBase.Reset.

These methods give us enough to start reasoning with facts, which leads us to the next topic... rules!

Rules

As for facts, we've changed the TValue type from *Yes, No, Unknown* to type ShortString for rules. Also, like facts, we again need to make sure a TRule is not a class that the end-user can play with. So, we've made the constructor protected and moved the properties from published to public. See Listing 7.

Note the CF property, which holds the certainty factor. Last time, we only used values 0 (for a condition) and 1 (for a conclusion), but this time we extend that to 0 (for a condition) and some value between 1 and 100 to indicate the certainty of the conclusion.

One other thing we need to consider is whether or not we need a case sensitive or case insensitive string compare (when comparing YES to Yes or yes for example). Let's deal with this later, and keep things case sensitive for now.

TRuleBase

One rule is seldom enough when reasoning. We often need an entire database of rules, called a rulebase. And while it would be unwise to have a component for every rule in the database, it might be helpful to have a component that encapsulates the entire rulebase: the TRuleBase component (Listing 8).

The design is similar to TFactBase: we're using delegation and maintain a field RuleTable of type TTable which points to the table which holds our rules. We need to create the table in our constructor and free it in our destructor. This is always the preferred way to create and free sub-components that are owned by our big mother component, which is also called a *SuperComponent* (for example by Mark "Mr.CDK" Miller) because the component itself consists of sub-components.

A RuleBase has a FactBase property of type TFactBase. Apart from the question of how we could assign such a value in the Object Inspector (hint: we need a property editor), the code is really simple. We also need a RuleBase filename property, which works exactly like the FactBase property of the TFactBase class: we just dissect the NewRuleBase into a DatabaseName and a TableName.

Again, we have a SetActive method and need to take care of loading a TFactBase from a stream file where the Active property is set to True. The methods Open and Close set the property Active to True and False respectively, which cause the SetActive method to be called. As for TFactBase we also need to be able to create a new empty rulebase by calling the NewRuleBase method.

Once we're done with a RuleBase session we can start again by resetting the rules (make sure none of them are "fired", so we can fire them again when needed). This ensures that we can perform many sessions without having to re-read the RuleBase, quite similar to the FactBase component again.

TFactBase and TRuleBase are all placed in one unit RULEBASE.PAS, together with the TFact and TRule classes. This means I can limit communication with the outside world and strengthen the internal coherence. TFactBase and TRuleBase are able to get to each others private parts, while these are shielded from the outside world. Add the fact that the constructors of TRule and TFact have been made protected and you'll find that the only way you can work with rules and facts is through the TRuleBase and TFactBase components. But that's more the programmer's interface. Let's first take a look at support for another end-user of these components: the run-time designer!

Open Tools API

Delphi offers a Tools API to allow programmers to extend the functionality of the Delphi IDE itself. There are four different Tool API interfaces, for Experts, Version Control Systems, Component Editors and Property Editors. They give us the ability to add to or enhance existing IDE features, and support component usage.

Property Editors

Property editors are extensions of the Delphi IDE. What does a property editor look like? Well, for starters, it is derived from a base class, TPropertyEditor, from which we need to override some methods in order to make things work our way. A TPropertyEditor edits a property of a component, or list of components, selected in the Object Inspector. The property editor is created based on the type of the property being edited as determined by the types registered by RegisterPropertyEditor.

The TPropertyEditor base class is defined in unit DSGNINTF.PAS and the methods we need to override for our purposes here are GetAttributes, GetValues and Edit.

GetAttributes determines the kind of property editor and its behaviour. There are three kinds of property editors (other than the default editbox type): a dropdown value list, a sub-property list and a dialog. GetAttributes returns a set of type TPropertyAttributes:

- ➢ paValueList: The property editor can return an enumerated list of values for the property. If GetValues calls Proc with values then this attribute should be set. This will cause the dropdown button to appear to the right of the property in the Object Inspector.
- ➢ paSubProperties: The property editor has sub-properties that will be displayed indented and below the current property in standard outline format. If GetProperties will generate property objects then this attribute should be set.
- ➢ paDialog: Indicates that the Edit method will bring up a dialog. This will cause the ... button to be displayed to the right of the property in the Object Inspector.
- ➢ paSortList: the Object Inspector will sort the list returned by GetValues (by name).
- ➢ paAutoUpdate: Causes the SetValue method to be called on each change made to the editor instead of after the change has been approved (eg the Caption property).
- ➢ paMultiSelect: Allows the property to be displayed when more than one component is selected. Some properties are not appropriate for multi-selection (eg the Name property).
- ➢ paReadOnly: Value is not allowed to change.

GetValue returns the string value of the property. By default this returns (unknown) and should be overridden to return the appropriate value. GetValues is called when paValueList is returned in GetAttributes. It should call the argument Proc for every value that is acceptable for this property.

Edit is called when the ... button is pressed or the property is double-clicked. This can, for example, bring up a dialog to allow the editing the property in some more meaningful fashion than by text (eg the Font property).

TFileName Property Editor

There are two special property types used by the TFactBase and TRuleBase components. The first is the name (and path) of the table that holds the factbase or rulebase, which is stored in a property of type TFileName. Instead of just typing the entire filename for these tables, it would be handy to be provided with a property editor that shows an OpenDialog instead. We did something very similar like this about a year ago for the TUuEncode and TUuDecode components way back in Issue 6 (February 1996), which describes how to write property editors.

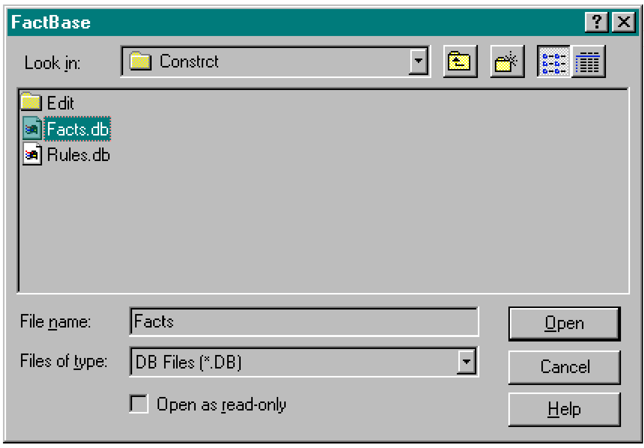
In Issue 6, we saw that while writing components is essentially a non-visual task (unless you're using one of these nifty Component Development Kits *[Ok, Bob, that's enough plugs! Editor]*), writing property editors is no different. We have to write a new unit by hand in the editor (see the listing for unit FileName below). We need to specify that we want a Dialog type of property editor, so we return [paDialog] in the GetAttributes function. Then we can do as we like in the Edit procedure, which in this case involves a TOpenDialog to let us find any existing file. See Listing 9.

Note that we call the GetName function of the property editor to get the name of the actual property for which we want to fire up the TOpenDialog. For the TFileName property called FactBase of the TFactBase component, this gives us the dialog shown in Figure 1.

In only a few lines of code we've written a TFileName property editor that will give great support at design time for all our components which use a property of type TFileName. This illustrates that property editors have an enormous potential for designers of Delphi components and applications.

TFactBase Property Editor

The FactBase property of the TRuleBase component adds a factbase to a rulebase. This is like the TTable



FactBase

Look in: Constrct

Edit
Facts.db
Rules.db

File name: Facts

Files of type: DB Files (*.DB)

□ Open as read-only

Figure 1

```
Type
  TFileNameProperty = class(TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
function TFileNameProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]
end {GetAttributes};
procedure TFileNameProperty.Edit;
begin
  with TOpenDialog.Create(nil) do
  try
    Title := GetName; { name of property as OpenDialog caption }
    Filename := GetValue;
    Filter := 'DB Files (*.DB)|*.DB';
    HelpContext := 0;
    Options := Options + [ofShowHelp, ofPathMustExist, ofFileMustExist];
    if Execute then SetValue(Filename)
  finally
    Free
  end
end {Edit};
```

Listing 9

```
Type
  TFactBaseProperty = class(TComponentProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValues(Proc: TGetStrProc); override;
  end;
function TFactBaseProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paValueList]
end {GetAttributes};
procedure TFactBaseProperty.GetValues(Proc: TGetStrProc);
var i: Integer;
begin
  with Designer.Form do begin
    for i:=0 to Pred(ComponentCount) do begin
      if (Components[i] is TFactBase) and (Components[i].Name <> '') then
        Proc(Components[i].Name)
    end
  end
end {GetValues};
```

Listing 10

TDataSource connection: every datasource (rulebase) must be connected to a table or query (factbase). Our connection works along the same lines.

First, we need to specify that this property editor consists of a list of values, returning [paValueList] in the GetAttributes function. Then, we need to return the value list itself, which contains the names of every component of type TFactBase which is on the same form.

Basically, we need to walk through the Components property of the form, use RTTI to see if they're of type TFactBase and if they are call the Proc method with their name to add them to the list of names to pick from. Unnamed components are not added to the list, of course. See Listing 10.

Now, it would be really interesting to try to create something similar to a Data Module for knowledge bases. A kind of *Knowledge Module* where you could put all your factbases and rulebases. For now, it's just an idea, but rest assured, we'll get back to the topic of Data Modules and the like in a future column...

Component Editors

Component Editors, the topic of *Under Construction* in Issue 8 (April 1996), are like property editors, in that they are used to enhance Delphi's IDE. Like property editors, they are basically derived from a single base class where some abstract methods need to be overridden and re-defined in order to give the component editor the desired behaviour. They are bound to a particular component type and are generally executed by a right mouse button click on the component when dropped onto a form. This way of activation is a bit different than property editors, but other than that, the process of writing your own component editor is essentially the same.

A component editor is created for each component that is selected in the form designer based on the component's type (see also GetComponentEditor and Register-ComponentEditor in the Delphi source file DSGNINTF.PAS). When the component is double-clicked the Edit method is called. When the context menu for the component is invoked, the GetVerbCount and GetVerb methods are called to build the menu. If one of the verbs are selected ExecuteVerb is called. Paste is called whenever the component is pasted to the clipboard. You only need to create a component editor if you wish to add verbs to the context menu, change the default double-click behaviour, or paste an additional clipboard format.

The class definition for the base class TComponentEditor can be found in DSGNINTF.PAS. There are six virtual methods which can be overridden. However, for this column we only need to focus on the Edit method, which is called when the user double-clicks the component. The component editor can bring up a dialog in response to this method, for example, or some kind of design expert.

TBaseForm

The TBaseForm is a simple form with a Table, DataSource, DBGrid and DBNavigator. It will open the Table and allow the user to enter new records, edit them, delete them, etc – the perfect simple way to allow editing of the contents of a factbase or rulebase. We'll use this as a template for the component editors.

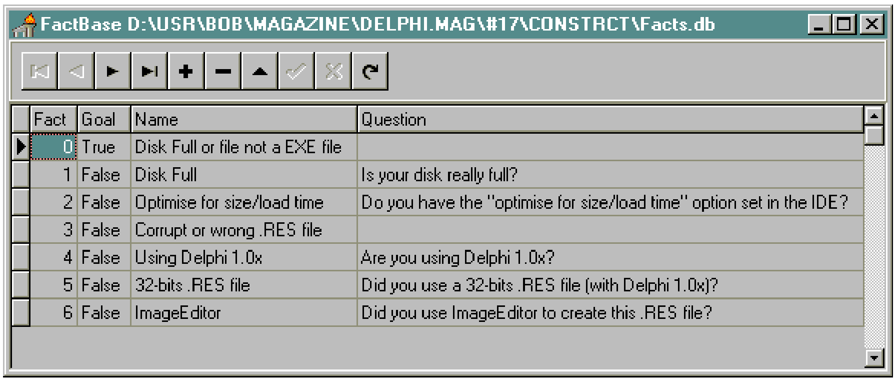
Since we have two components it's only logical we should also write two component editors. They are very much alike, however, as we will see shortly.

TFactBaseComponentEditor

This component editor should allow us to edit the facts in the factbase. For this, we need to have the FactBase filename property (which is split into the DatabaseName and TableName). The FactBase does not have to be open, since this component editor will work with the hidden table itself and not the facts loaded in memory. We only need to override the Edit method of the TComponentEditor class, then create and show the TBaseForm, while making sure the Table on that form is pointing to the correct FactBase. See Listing 11.

The Caption is set to remind us which FactBase we're editing. See Figure 2.

Note that we're editing the FactBase on disk. So, if we've opened it before, we will have facts loaded in memory. These are not automatically updated when we close the component editor. In fact, the facts in memory will still be the old ones. We need to close and re-open the FactBase to update the facts in memory. I could've made this an automatic (and invisible) step, but I can also think of a situation where you would want to update the facts in the FactBase on disk while still working (reasoning) with the current FactBase (for example if you don't want to close the FactBase and break off the current reasoning path, but might want to fix some bug in a fact anyway).



FactBase D:\USR\BOB\MAGAZINE\DELPHI.MAG\#17\CONSTRCT\Facts.db

Fact	Goal	Name	Question
0	True	Disk Full or file not a EXE file	
1	False	Disk Full	Is your disk really full?
2	False	Optimise for size/load time	Do you have the "optimise for size/load time" option set in the IDE?
3	False	Corrupt or wrong .RES file	
4	False	Using Delphi 1.0x	Are you using Delphi 1.0x?
5	False	32-bits .RES file	Did you use a 32-bits .RES file (with Delphi 1.0x)?
6	False	ImageEditor	Did you use ImageEditor to create this .RES file?

➤ *Figure 2*

```
Type
  TFactBaseComponentEditor = class(TComponentEditor)
  public
    procedure Edit; override;
  end;
procedure TFactBaseComponentEditor.Edit;
begin
  with TBaseForm.Create(nil) do
  try
    Caption := 'FactBase '+(Component AS TFactBase).FactBase;
    Table1.DataBaseName := (Component AS TFactBase).FactTable.DataBaseName;
    Table1.TableName := (Component AS TFactBase).FactTable.TableName;
    ShowModal
  finally
    Free
  end
end {Edit};
```

➤ *Listing 11*

```
Type
  TRuleBaseComponentEditor = class(TComponentEditor)
  public
    procedure Edit; override;
  end;
procedure TRuleBaseComponentEditor.Edit;
begin
  with TBaseForm.Create(nil) do
  try
    Caption := 'RuleBase '+(Component AS TRuleBase).RuleBase;
    Table1.DataBaseName := (Component AS TRuleBase).RuleTable.DataBaseName;
    Table1.TableName := (Component AS TRuleBase).RuleTable.TableName;
    ShowModal
  finally
    Free
  end
end {Edit};
```

➤ *Listing 12*

TRuleBaseComponentEditor

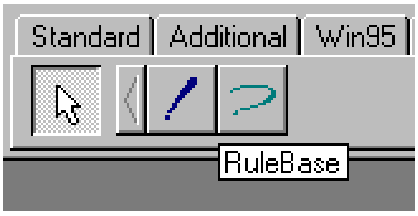
A similar thing needs to be done for the TRuleBase component, for which we'll write a TRuleBaseComponentEditor. Again, we only need to override the edit method of the TComponentEditor class (Listing 12).

Using RTTI, we could have written just one component editor for both components. In fact, I have done so, but I leave the code details as an exercise for the reader (don't be afraid to e-mail me if you can't get it to work).

Component Bitmaps

Now we're almost done with the design-time look and feel for the two inference engine components. All we need are two nice bitmaps for the factbase and rulebase. This is where Bolesian comes in again. Our logo (Figure 3) contains an integrated question mark (green) and exclamation mark (blue). This means something like *for every (client) question, we have an answer*. The blue exclamation mark could also stand for facts (something we know), while the green question mark could stand for a rule (something that needs to be

➤ *Figure 3*



➤ *Figure 4*

```
procedure Register;
begin
  { components }
  RegisterComponents('Dr.Bob', [TFactBase, TRuleBase]);
  { property editors }
  RegisterPropertyEditor(TypeInfo(TFileName), TFactBase, 'FactBase',
                         TFileNameProperty);
  RegisterPropertyEditor(TypeInfo(TFileName), TRuleBase, 'RuleBase',
                         TFileNameProperty);
  RegisterPropertyEditor(TypeInfo(TFactBase), TRuleBase, 'FactBase',
                         TFactBaseProperty);
  { component editors }
  RegisterComponentEditor(TFactBase, TFactBaseComponentEditor);
  RegisterComponentEditor(TRuleBase, TRuleBaseComponentEditor)
end;
```

➤ *Listing 13*

```
function TRuleBase.TestRule(RuleNr: Integer): Boolean;
var i: Integer;
begin
  Result := True;
  for i:=0 to Pred(FNumRule) do
    if (Rules[i].Rule = RuleNr) and (Rules[i].CF = 0) then { check }
      Result := Result AND
        (FFactBase.Facts[Rules[i].Fact].Value = Rules[i].Value)
       { NOTE: we need to compare two strings case-insensitive here... }
end {TestRule};
```

➤ *Listing 14*

executed and proven). I decided to use these as component bitmaps, for which they have to be 18x18 pixels and in 16 colours. We also need to give them the resource names (all uppercase) of the corresponding components: TFACTBASE for the exclamation mark and TRULEBASE for the question mark.

Installation

Installing the two components also consists of installing the two supporting property editors and the two component editors, which makes for a relative big Register procedure. First we register the two components in one statement on the Dr.Bob tab of the component palette. Then, we need to register the two property editors, which actually takes three statements (the TFileNameProperty editor is installed twice: once for the FactBase property of a TFactBase and then for a RuleBase property of a TRuleBase). When registering a property editor, we need to supply type information for the property, the type of component that has this property, the name of the property and finally the type of property editor itself.

Registering a component editor is much simpler: we just need two parameters to a function called RegisterComponentEditor. The first is the name (type) of the relevant component (TDialog in our case), the second parameter is the type of the component editor itself (TDialogEditor). See Listing 13.

Figure 4 shows the two components on the component palette in the Dr.Bob tab. Drop them on a form and we can test the property editors (for example to assign FactBase1 to RuleBase1.FactBase) and component editors (to fill in some new facts or rules).

Inference Engine

Last month, we wrote three supporting routines for the forward and backward chaining algorithms: TestRule, to see if the conditions of a rule were satisfied; FireRule, to fire a rule and add the conclusion to the fact set; and Conclude, to find any rule that could be used to prove a certain fact.

These routines, and forward and backward chaining itself, were simple when we only had to worry about three possible values: Yes (true), No (false) or Unknown. Now, when dealing with string values, we need to check every fact against the required string value in the rule. If we use = for this, we get a case sensitive compare and if we don't want this we should make sure to use the CompareText function instead (see the online Help). The three methods TestRule, FireRule and Conclude are now members of TRuleBase and have to be adjusted to find the Facts using the FFactBase field of this component. Furthermore, we've modified TestRule to compare strings instead of just checking for Yes values. See Listing 14.

The other two routines didn't change that much (full source code is on the disk, of course).

Backward Chaining

Forward chaining essentially remains the same, but we need to change some parts of the backward chaining algorithm. Last time, we noted that we could stop investigating a certain rule as soon as one of its conditions turned out to be false. This is no longer the case, since we no longer simply check for Yes values (the rulebase still contains only Yes values but this is not important, since the components and algorithm are now capable of checking for any value). We could modify the question *"Are you using Delphi 1?"* to one that would ask for the specific version of Delphi being used, in which case 1.x would be a detailed answer enough. In this version, though, I'm still using a simple Yes/No messagebox (but you're free of course to extend this to use a new kind of dialog or form).

What's more interesting right now is the fact that we need to check for the existence of both the FactBase and RuleBase. Furthermore, both must be Active (opened), which means the collection of rules and facts is available for our inference engine to use. We'll just raise an exception if any of these conditions is false. See Listing 15.

Note that the Backwards method is writing information to the standard output. This will only work if you've checked the CONSOLE option for Delphi 2.x applications (or are using the {$APPTYPE CONSOLE} compiler directive), or if you've included the WinCrt unit in your uses clause for Delphi 1.x applications. While they are not a real part of the communication with the consulting user, I've still left them in for tracing purposes: the output in the console or WinCRT window will show you how "deep" the backward chaining algorithm is and what path the entire consultation followed. Can be very interesting to watch (and is helpful when you are debugging rulebases as well).

Certainty Factors

When dealing with certainty factors, we need to realise first what it means to use uncertainty in rule based systems. Normally, a rule would conclude something with a certainty factor of 100% (like *IF your disk is full THEN you don't have any space left*). However, sometimes a rule can conclude something with a lesser certainty (like *IF you're using ImageEditor THEN your resource file might get corrupt*, with a chance of less than 1%).

Of course, this would also mean that a fact is no longer certain for 100%, so we need to introduce a field CF inside the TFact class as well, which would in turn influence the certainty of a rule. If a rule says that if condition A is true then conclusion B is a fact with certainty 60%, we need to check the certainty of condition A as well. If A is a fact with CF 40%, then we can derive B, but only with a CF of 40% * 60% which is 24%. The algorithm is in fact dealing with certainty factors as if they were probabilities, which is one of the ways to treat uncertainty in rules. Other algorithms, such as the Bayesian approach, are more complex, and relate more toward a fuzzy logic approach (which is a topic that can wait for another time).

We can conclude by stating that certainty factors can be implemented by adding a CF field to the TFact class and enhancing the FireRule method to calculate a CF value for every new fact. This should be enough for now.

Conclusion

Using the TFactBase and TRuleBase components we can build a knowledge-based application really quickly. When executing the application, we get a console (or WinCRT) window next to our application window with the knowledge trace information.

We haven't been able to extend the facts to include ranges of values or include an explanation facility – the so-called "why" function that explains to the user why (or how) a certain conclusion has been reached. These features will be left as exercises for the reader (hint: why/how information can be realised using a history of the trace path). A full-blown version of the RuleBase unit will be available on my homepage shortly (see below), ready to be used with Delphi 1 or higher, and C++ Builder when it's available.

Next Time

We've seen enough facts and rules for a while now. Next time, we'll get back to visual component building and touch on some porting issues whilst we're at it. Stay tuned and don't forget to make a backup of your Component Library whenever installing something new or experimental...

Bob Swart (home.pi.net/~drbob/) is a professional knowledge engineer and technical consultant using Delphi and C++ for Bolesian (www.bolesian.com), freelance author and co-author of *The Revolutionary Guide to Delphi 2*. He is co-working on a new book about Delphi and the internet. In his spare time, Bob likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 2.5-year old son Erik Mark Pascal and his newborn daughter Natasha Louise Delphine.

➤ *Listing 15*

```
procedure TRuleBase.Backwards(Goal: Integer);
Const Depth: Word = 0;
var i,j: Integer;
begin
  if (FFactBase = nil) then
    raise Exception.Create('no FactBase');
  if not FFactBase.Active then
    raise Exception.Create('FactBase not open');
  if not Active then
    raise Exception.Create('RuleBase not open');
  Inc(Depth);
  writeln(' ':Depth,Goal);
  i := 0;
  while i <= RuleMax do begin { all rules }
    if Conclude(i,Goal) then begin
      if TestRule(i) then
        FireRule(i)
      else begin { infer or ask }
        j := 0;
        while j < NumRule do begin
          if (Rules[j].Rule = i) and (Rules[j].CF = 0) and
            (FFactBase.Facts[Rules[j].Fact].Value =
            'unknown') then begin
            Backwards(Rules[j].Fact); { infer }
            if TestRule(i) then
              j := NumRule
            else begin { ask }
              if FFactBase.Facts[Rules[j].Fact].Question <>
                '' then begin
                writeln(' ':Depth,
                  FFactBase.Facts[Rules[j].Fact].Question);
                if MessageDlg(
                  FFactBase.Facts[Rules[j].Fact].Question,
                    mtConfirmation, [mbYes,mbNo],0) =
                  mrYes then
                  FFactBase.Facts[Rules[j].Fact].Value :=
                    'Yes'
                else
                  FFactBase.Facts[Rules[j].Fact].Value
                    := 'No'
              end;
              if TestRule(i) then
                j := NumRule
            end
          end;
          Inc(j)
        end;
        if TestRule(i) then begin
          FireRule(i);
          i := RuleMax
        end;
      end
    end;
    Inc(i)
  end;
  Dec(Depth);
  if Depth = 0 then begin
    { final goal proven? }
    writeln;
    writeln(FFactBase.Facts[Goal].Name,': ',
            FFactBase.Facts[Goal].Value);
    ShowMessage(FFactBase.Facts[Goal].Name + #13 +
                FFactBase.Facts[Goal].Value)
  end
end;
```