Better Calculated Fields

by Rohit Gupta

Calculated fields are potentially very useful. However, the way Inprise has implemented them is not how I wanted to use them: I needed calculated fields that I could read and write to when I wanted, without being restricted to using an `OnCalc` event handler or having to put the dataset into edit/insert mode. With the standard Delphi implementation, this is just not possible. Also, the overhead of recalculating all the calculated fields when any field value changes is too much for me.

One way to solve this would have been to derive components, but this would have required replacements for `DataSource`, `Table`, `Query`, `Fields`, etc. I kept dithering until I came across Stephen Posey's article *Interposer Classes*, in Issue 33 (May 1998).

Interposer Classes

So what are interposer classes (as Stephen has dubbed them)? The technique has been useful in C for overriding standard Microsoft library functions with your own ones by coaxing the linker to use your routines of the same name instead of the standard ones.

In Delphi, the technique works by including a unit containing replacement classes in the right place in the `uses` clause, so that the linker links your classes in instead of the Delphi (or third party) supplied ones.

➤ *Listing 1*

```
unit CalcFlds;
interface
uses
  Classes, DB;
type
  TStringField =
   class(DB.TStringField)
  protected
    function GetCanModify :
      boolean; override;
  end;
implementation
function TStringField.GetCanModify:
  boolean;
begin
  Result := not ReadOnly;
end;
end.
```

Fields

The first task is to ensure that the fields can be modified via edit controls. Having scoured the VCL source I am unable to find which bit of code decides that the calculated fields return `CanModify` as `False`. Therefore, I have taken the brute force approach of overriding `CanModify`, as illustrated in Listing 1. This shows how `TStringField` has been overridden. Note that the new class has the same name as the original. To ensure that the compiler knows which is which, it is essential to prefix the original one with the unit name it belongs to, hence `DB.TStringField`. The `GetCanModify` now returns `True` if `ReadOnly` is `False`.

All the standard field classes need to be modified or interposed in this way: look at the unit `CalcFlds` on this month's disk.

The magic here works on every unit that has `CalcFlds` in the `uses` clause. It is essential that `CalcFlds` appears after `DB` in the `uses` clause. All `TFields` defined in that unit are created with this new class, therefore all polymorphic properties are maintained. That is, if another standard Delphi unit calls `GetCanModify`, the interposed method is automatically executed. That's right: you have dynamically modified a base class's method.

Table

We need some method to initialise the calculated fields on `DataChange`. Unfortunately, `TDataSource` does not define virtual methods that we can use. But `TTable` does.

Listing 2 illustrates how `DoAfterScroll` has been overridden. First, `SetTempState` is called. This returns the current state and sets it to `NewValue`, which is an internal state for such purposes. Next, the inherited `DoAfterScroll` is called. Finally, the original state is restored.

There is one other caveat. Somewhere in the depths of VCL, it does not fire up the datachanged message/event for calculated fields. This means that linked edit controls are not updated with the new values. The easiest thing to do is to iterate through all fields and for each calculated field, fire off the appropriate event. Note that if you are using other `DataSets` (eg `TQuery` or `TwwTable`), you need to override them instead of `TTable`.

Remember that the `CalcTbls` unit must be in your `uses` clause after `DBTables`. Put the initialisation code for the calculated fields in the `AfterScroll` event.

Databases

I use Btrieve and Titan for all my projects and I found that I needed to override `ClearCalcFields` to prevent calculated fields from being cleared every time any field changed in value. This is not required for dBase using the BDE. I have no idea what happens with Paradox tables. It does no harm to prevent the clearing. For completeness, the `AutoCalcFields` flag is used to control this. If you find your calculated fields disappearing, set `AutoCalcFields` to `False`.

Example

The example supplied on the disk is a trivial one to show how one can keep the pre-tax price of a bill item and the tax in the database. And you can have the `Price_with_tax` field available. The example shows how to modify any of the three fields and one of the others is automatically calculated. This would be a tad impossible with normal Delphi behaviour.

To run the example, first load the projcalc.dpr in Delphi. Open unitcalc.pas and click on the `Table` component on the form. Change the `DatabaseName` property in the Object Inspector to the full pathname of the directory that you have placed all the files into. Now set `Active` to `True`, compile and run.

To see what standard Delphi would do, just remove `CalcFlds` and `CalcTbls` from the `uses` clause and run again.

```
unit CalcTbls;
interface
uses Classes, DB, DBTables;
type
  TTable = class(DBTables.TTable)
  protected
    procedure DoAfterScroll; override;
    procedure ClearCalcFields(Buffer : PChar); override;
  end;
implementation
procedure TTable.DoAfterScroll;
var OldState : TDataSetState;
    I        : smallint;
begin
  OldState := SetTempState (dsNewValue);
  try
    inherited;
  finally
    RestoreState (OldState);
    for I := 0 to FieldCount-1 do
      if Fields [I].FieldKind = fkCalculated then
        DataEvent(deFieldChange,longint(Fields [I]));
  end;
end;
procedure TTable.ClearCalcFields (Buffer : PChar);
begin
  // prevents calculated fields from being wiped if reqd
  if AutoCalcFields then inherited; // may need to call inherited on datachange
end;
end.
```

➤ *Listing 2*

Conclusion

Stephen Posey did an excellent job of discovering interposer classes. Hopefully, I have added to that with a practical example where they are invaluable. Please note that all my testing has been on Delphi 3, but there is no reason for this approach to not work on Delphi 1, 2 and 4.

As an aside, my requirement was to have a database table with variable numbers and types of fields that would be used on different data entry forms. These forms would multiply like rabbits over time and it would be impractical to have separate tables for each. Another solution would have been to use normalised tables: that is, a header table slaved to another table of variants. This would mean that each form would typically pull in 100 to 300 records from the slave table. The thought was rather unpalatable.

What I have done, using interposer classes, is to pack a variable number of fields (of varying types) into one variable length memo field. Each form creates calculated fields of types and quantities required and the new table class automatically links them up to the data and does the packing and unpacking. If I can find some spare time, I may describe this in another article.

Rohit Gupta works for Computer Fanatics Ltd in New Zealand, where he deals mostly with Veterinary Hospital, Hair Dressing and Beauty Salon systems. At home he has two children and 10 acres to look after. Email him at rohit@cfl.co.nz