# The Observer Pattern Observed

*by Tomasz Stanczak*

In Issue 51 (November 1999) I found an interesting article about the Observer Pattern, by Peter Hinrichsen. We faced similar needs in our last software development contract and solved them using a similar approach, so I thought I would share our experience in the areas Peter did not cover in his article.

At the end of the article Peter describes a 'push' model, and states that, while being more efficient, it is more challenging to implement. Well, this is true, but it is not that difficult after all, and here is why.

Our contract was for a quite a big application, with hundreds of forms and classes, and hundreds of thousands of lines of code. The observer pattern was used throughout the application to achieve instant communication between the various layers of the application. A 'pull' model proved to be too inefficient for our needs, so we decided to implement a 'push' model.

Peter states that this requires a tighter coupling between subject and observers. We have managed to implement a model that is loosely coupled, very flexible, and that notifies only about items which have really changed.

If you think about Peter's example implementation, it is obvious that the method `DataToObserver` must be changed to achieve this. It must have parameters describing what has been changed. A simple solution using numerical constants has the disadvantage of being not easily readable and is also exposed to the risk of duplicating values (that's actually the same pattern which is used by Windows messages). An enumerated type doesn't allow encapsulation, such a type must either be visible application-wide, so different subjects would require to share it and it wouldn't be possible to extend subjects without extending this common enumeration type, or the declaration of the `DataToObserver` method would have to be changed with each subject, which in turn means changing the abstract base interface.

While analyzing the possible solutions, we came to the following conclusion: it's all about objects, so why not use objects to carry information? Let's define a class `TSubjectMessage` as a base class. If we change the method mentioned aove to:

```
DataToObserver(
  Message: TSubjectMessage)
```

we would allow observers to react only to changes that are relevant for them. It is not the end of the required modifications, of course: it can't be called by a parameterless `UpdateObservers` method, we would have to modify the subject to include setter methods for all changes that in turn would call `UpdateObservers` with the same parameter, that would then be used to notify observers.

Now to have different messages we would just inherit from `TSubjectMessage`. The benefits are obvious: it makes a black-box approach possible. Adding a new subject that uses new messages is just a matter of defining new `TSubjectMessage` descendants for new notification types. No source code changes to the original abstract framework are required.

While using objects to carry information allows much more flexibility, we could extend the base class to accept parameters, like Delphi's `TParams` class (see Listing 1).

Now you are able to send notifications not only about items changed, but the notification object could contain useful information itself. Observers are able to recognise what has changed and how, and so refresh only when required. And all that without the subject knowing anything about attached observers, absolutely loosely coupled.

➤ *Listing 1*

```
procedure TPortfolio.SetPrice(const StockName: string; Value: Real);
var
  i: integer;
begin
  for i:=0 to FList.Count-1 do begin
    if CompareText(TStockPrice(FList).StockName,StockName) = 0 then begin
      if TStockPrice(FList).Price <> Value then begin
        FPrice := Value;
        Msg := TStockChangeMessage.Create(nil);
        Msg.CreateParam(ptString,'StockName');
        Msg.Params['StockName'].AsString := FStockName;
        Msg.CreateParam(ptReal,'Price');
        Msg. Params['Price'].AsReal := Value;
        UpdateObservers(Msg);
      end;
      break;
    end;
  end;
end;
procedure TSubjectAbstract.UpdateObservers(Message: TSubjectMessage);
var
  i: integer;
begin
  try
    for i:=0 to FObservers.Count-1 do
      TObserverAbstract(FObservers[i]).DataToObserver(Message);
  finally
    If Message.Owner=nil then
      Message.Free;
  end;
end;
procedure TObserverBarChart.DataToObserver(Message: TSubjectMessage);
begin
  if (Message is TStockChangeMessage) and
    Displaying(Message.Params['StockName'].AsString) then
    RefreshStock(Message.Params['StockName'].AsString,
      Message.Params['Price'].AsReal);
end;
```

A good example of the use of this approach might be a drill down action. Imagine having a grid with data entered in different ways: some manually from a data entry form, other data as part of an automatic data import through email. Double-clicking on a grid row may send a message with the row primary key as a message parameter. Only the observer responsible for this very row would react and come to the foreground.

I admit that the approach requires more code than the solution of 'pull' observers, but it is much more efficient and flexible. There is, of course, room for improvements, such as implementation of the `BeginUpdate`/`EndUpdate` method pair within the `TSubjectAbstract`. It enables us to defer notifications when you want to make more changes at once and only notify the observers at the very end.

The implementation could be wrapped up as VCL components, so it is not necessary for the subject to inherit from `TForm` and it can be used not only on forms or frames but within any other kind of object, with appropriate notification. This is, in fact, the way we have implemented the pattern in our framework.

Furthermore, there is a place for an implementation sending the notifications within a background thread, instead of using timers. Well, OK, this is much more challenging (not because of the threading itself, but because of the synchronisation issues). However, maybe readers could provide a solution from their experience? Do email me if you have anything new to add, or any comments on the solution we used.

Tomasz Stanczak (email tomasz@rtsoftware.com) is one of the proprietors of r&t software, a German-Polish company specialising in custom software design and development in Delphi. A significant amount of thought on this topic has come from Krzysztof Janiszewski, his partner (email krzysztof@rtsoftware.com).