The Observer Pattern

Making the most of design patterns in Delphi

by Peter Hinrichsen

An excellent indication of the quality, efficiency and cost effectiveness of a software project's internal design and structure is the amount of code that can be reused for subsequent projects. The use of design patterns is an excellent way to develop code that can be transferred from project to project.

In my early days of developing with Delphi, I had a single unit of utilities (string manipulation, registry processing, custom dialogs, etc), and some custom components, which I took from project to project. I would reuse functions like `tiMixedCase`, that converts a string into upper and lower case (Like This), but when it came to database access, business objects and GUI development, I had to start from scratch every time. To maximise the reuse of code, what was needed was an application framework that would be versatile enough to be applied to any project with a small amount of customisation.

About a year ago, I had my first introduction to design patterns and very quickly learnt that there was a way to develop an application framework, which could be reused and tailored to match the needs of most projects. Two patterns that can now be found in nearly all my work are the singleton and factory patterns; these have both been discussed in past issues of *The Delphi Magazine*.

In this article, we are going to discuss the observer pattern, as introduced in the 'Gang of Four' book (hereafter GoF, see the reference at the end of the article).

The observer pattern provides a powerful technique for decoupling the presentation layer of an application from the business objects or the data modelled by the application, while still managing to keep the separate components of the app consistent as the data changes.

The Intent Of The Observer

To quote from GoF, the intent of the observer pattern is to 'define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically.' Figure 1 (taken from GoF) is the best illustration of this that I have seen.

The diagram shows a business object model (called the subject), which contains some data we want to view in a variety of ways. There are three observers, each displaying the content of the subject in a different way. One observer enables us to browse the data in a grid or listview, the other two represent the data as a pie graph and bar graph respectively.

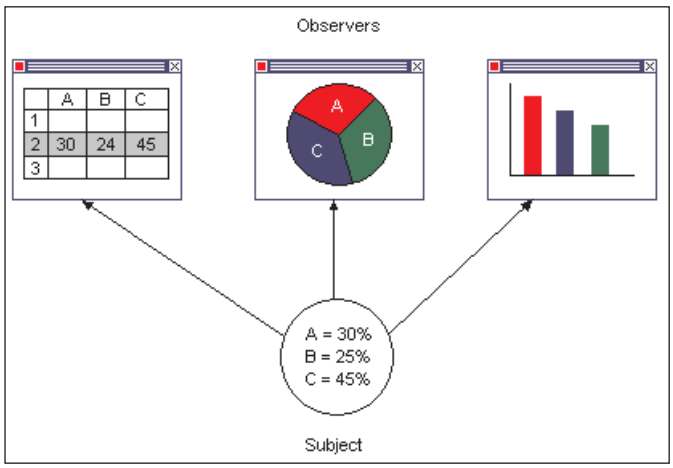
Motivation

There are many ways to achieve this functionality in Delphi. An obvious option is to create three forms, one with a `TDBGrid` and two with `TDBChart` components. Next, drop a `TDataSource` and `TTable` or `TQuery` on a `TDataModule`, then link all the elements together. The three forms are all connected to the same `TDataSource`, which is in turn attached to a file-based table or the result of an SQL query against a DBMS. The `TDataSource` is responsible for keeping all the pieces synchronized, a task that it does very well.

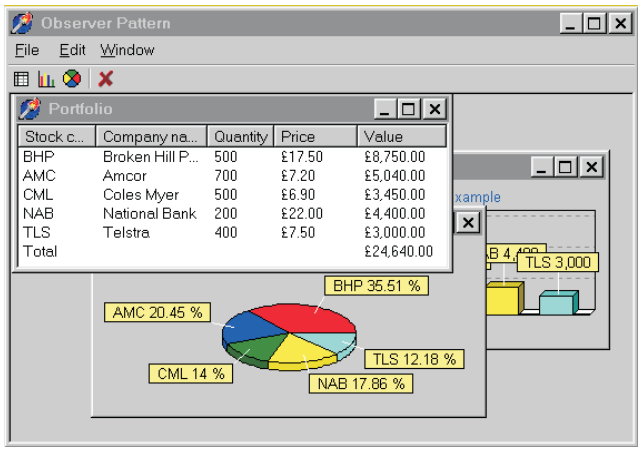
There are, however, several problems with this approach. As our application grows we find maintenance difficult because of the tight coupling between the GUI and the database. We have also built in a dependency on data-aware controls, which are good for prototypes and low volume applications, but result in an application which is difficult to maintain and causes excessive network traffic when used in large, client/server projects.

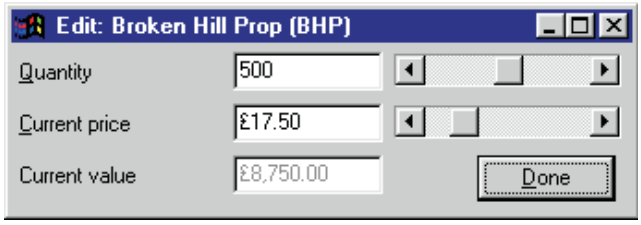
The solution is to partition the application so that there is a business object model between the database and the user interface. This solution reduces dependencies, but also means we no longer have access to the `TDataSource` to manage synchronisation of the various forms when changes are made to the underlying data.

The answer to *this* problem lies in the observer pattern. When the observer pattern has been implemented, the observers will always be synchronised with the subject. The subject will have no knowledge of the specific observers, except that they descend from an abstract observer, and hence all have certain common elements in the interface. The observers will know about the subject (after all, they must if they are going to display its data) but they will not know anything about each other. We will be able to make changes to



➤ *Figure 1*





➤ *Left, Top: Figure 2*

➤ *Left, Bottom: Figure 3*

The example shows an MDI application with three read-only observers (I call them read-only because they can see the data but can't change it). A fourth observer, shown in Figure 3, will read the subject as well as update it, as its slider bars are moved (the form to edit a record can be loaded by double clicking the grid). I found this application good fun to play with, as it is possible to have dozens of forms on the screen, all updating as you use the mouse to adjust a slide bar.

The UML for the business object model we will 'observe' is shown in Figure 4. Our application contains a single `TPortfolio` object, which is really just a holder for a list of `TStockTrans` objects. The `TStockTrans` class has the properties `StockCode`, `StockName`, `Price` and `Quantity`, as well as the derived property `Value`, which is simply `Price*Quantity`. These two objects combine to form an OO replacement for the rows and columns of a `TTable` or `TQuery`.

If you are new to UML, you will need to know that the arrow with the ◆ and * tells us that the `TPortFolio` owns 0..n `TStockTrans` objects. Public properties and methods are shown in the lower box for each class.

A Simple Observer

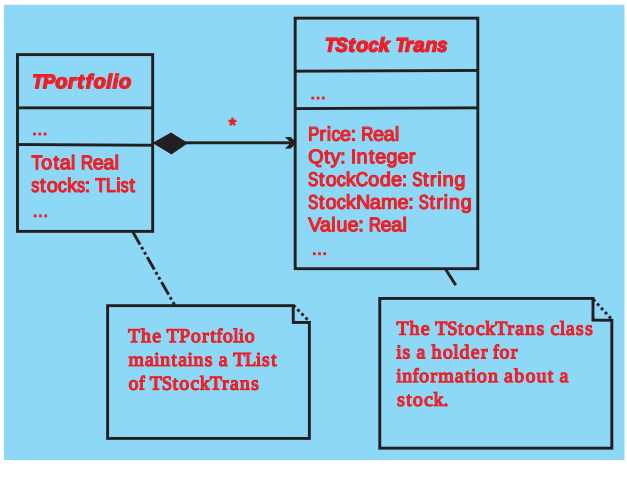
The UML in Figure 5 gives an overview of the framework of the application we are studying.

There is a `TSubjectAbstract` class and a `TObserverAbstract` class, which have a dependency between them that is shown by the two arrows joining the two classes. Descending from these are the concrete `TPortfolio` and the `TObservers`.

The `TPortfolio` object is created by the application as a singleton, so there is only one copy in existence. The four observers are created as MDI children by a button click in the main form. Some more detailed UML for the `TSubjectAbstract` and `TObserverAbstract` is shown in Figure 6. We will now look at the implementation of the `TSubjectAbstract`, then the `TObserverAbstract`. The concrete observers and the application framework that will tie all the pieces together will follow this.

The interface of the `TSubjectAbstract` is shown in Listing 1. There is a private `TList` called `FObservers` used to hold pointers to all the observers currently interested in this subject. There are three methods of interest: `AttachObserver`, `DetachObserver` and `UpdateObservers`.

`AttachObserver` takes a single `TObserverAbstract` as a parameter and adds it to the `TList` (after checking to make sure it has not already been added). I added an `Assert` to check that the observer was not already attached to help in debugging. Delphi will remove the `Assertion` for a release build of the project by clearing the `Assertions` checkbox on the `compiler` page of the `project | options` dialog. The



➤ *Figure 4*

Solving A Business Problem

Sometimes the best way to discuss a new concept is to put it into the context of a problem we are familiar with, so we will use a small application to track the value of a share portfolio as a means to discuss the observer.

Our application will show the current share price of five blue chip Australian companies. The number of shares of each company in our portfolio will be shown, along with the calculated value of the shares and the total value of the portfolio. The user is able to adjust the amount of each stock he/she is holding to balance the portfolio between the different industry sectors. This application is shown in Figure 2.

the observers, or add and remove them with no risk of breaking any existing code.

The observers are also able to make changes to the data contained in the subject, then the subject will notify all the other observers that a change has been made and that they should resynchronise themselves with the subject. This may seem like a lot of work, just to reproduce the functionality of the `TDataSource`, but if there are good reasons to avoid data-aware controls, it is worth the effort.

implementation of AttachObserver is shown in Listing 2.

DetachObserver also takes a TObserverAbstract as a parameter and removes the pointer to the observer from the list. The implementation is shown in Listing 3.

UpdateObservers scans the TList and calls the DataToObserver method for each observer in the list. This is shown in Listing 4.

That is about all there is to TSubjectAbstract class, so we'll look at the TObserverAbstract class next. The class interface is shown in Listing 5. TObserverAbsract descends from TForm and has some code in the form's OnClose, OnDestroy and OnShow events.

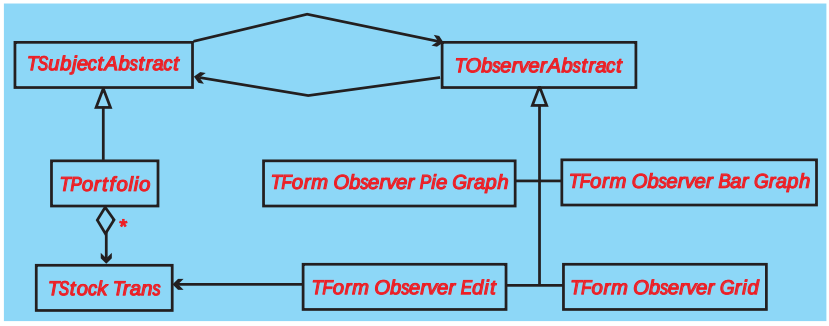
Because this is an MDI application, and we have no control over how many instances of a form the user may want, or when the forms will be closed, we set the Action parameter in FormClose to caFree so the form self-destructs when it is closed. FormShow contains a call to DataToObserver in order to ensure that the form contains the latest view of the data when it becomes visible. FormDestroy contains the line Subject.DetachObserver which removes it from the list of observers the subject must notify of updates.

There is the property Subject with a corresponding variable called FSubject and the SetSubject method that holds a pointer to the subject we are interested in observing. SetSubject contains the usual FSubject := Value line as well as a call to Subject.AttachObserver(self). SetSubject is shown in Listing 6.

The last method in TObserverAbstract that we must know about is the virtual abstract procedure DataToObserver. A virtual method is one which can be overridden and extended by a child class and an abstract method has an interface but no implementation (or code). A virtual abstract *must* be implemented in the child class otherwise the compiler will complain and an exception will be raised at runtime. DataToObserver contains the code to read the subject into the observer for display.

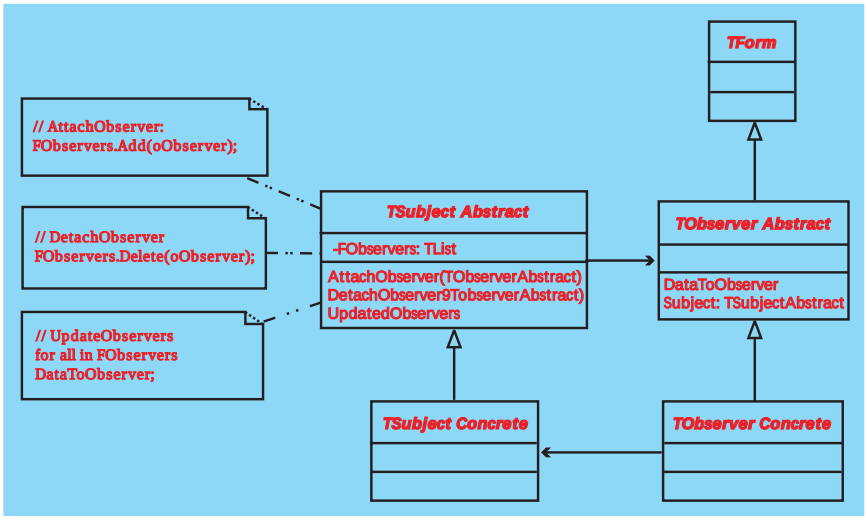
Because we have four observers, we must write four separate implementations of DataToObserver. Two of these will copy data into a TChart for display; the code for this is in Listing 7.

Notice how we must typecast (Subject as TPortfolio). We could use the alternative (and, I think,



➤ *Above: Figure 5*

➤ *Below: Figure 6*




```
TSubjectAbstract = class( TObject )
private
  FObservers : TList ;
protected
public
  Constructor Create ;
  Destructor  Destroy ; override ;
  Procedure   AttachObserver(Sender :TObserverAbstract);
  Procedure   DetachObserver(Sender :TObserverAbstract);
  Procedure   UpdateObservers ;
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
procedure TSubjectAbstract.AttachObserver(Sender: TObserverAbstract);
Var i : integer ;
begin
  i := FObservers.IndexOf( Sender ) ;
  Assert( i = -1, 'Observer already attached' ) ;
  FObservers.Add( Sender ) ;
end;
```

```
procedure TSubjectAbstract.DetachObserver(Sender: TObserverAbstract);
Var i : integer ;
begin
  i := FObservers.IndexOf( Sender ) ;
  Assert( i <> -1, 'Observer not attached' ) ;
  FObservers.Delete( i ) ;
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
procedure TSubjectAbstract.UpdateObservers;
var i : integer ;
begin
  for i := 0 to FObservers.Count - 1 do
    TObserverAbstract( FObservers[i] ).DataToObserver ;
end;
```

more readable) syntax `TPortfolio(Subject)` but if, for some reason, `Subject` can not be typecast as `TPortFolio`, then `TPortfolio(Subject)` will cause an access violation giving us very little information about what the error was or where it occurred. The typecast `(Subject as TPortfolio)` will raise an 'invalid type conversion' exception, which has more meaning while we are debugging than a big, bad, access violation.

The listview observer has some equally simple code in the `DataToObserver` method, which is shown in Listing 8.

As you can see, in its simplest form, the observer pattern is very easy to implement. In some situations, however, the observer may also need to update the subject, as required in the form in Figure 3. This form features three `TEdits` (which you can't edit directly because their `enabled` property is set to `false`) and two `TScrollBars` that let the user change the value of the `Quantity` or `Current Price` fields.

Before learning about the observer pattern (and its close cousin, the model-view-controller) I would have added code to the `TScrollBar`'s `OnChange` event to update the value in the corresponding `TEdit`. This new value would have been written to the subject when the form's `OK` button was clicked. As one of the aims of the observer is to enable many views of some dynamically changing data, the `TScrollBar`'s `OnChange` event causes the subject to be updated, then calls the subject's `UpdateObservers` method to ensure consistencey between all the observers. Listing 9 shows both the edit form's `DataToObserver` method, as well as one of the `TScrollBar`'s `OnChange` events.

Other Considerations

Once again, the observer pattern is very simple to implement at this level. However, when you start thinking more deeply about the pattern, there are several questions that require further attention.

```
TObserverAbstract = class(TForm)
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  procedure FormDestroy(Sender: TObject);
  procedure FormShow(Sender: TObject);
private
  FSubject : TSubjectAbstract ;
  procedure SetSubject(const Value: TSubjectAbstract);
public
  procedure DataToObserver ; virtual ; abstract ;
  property  Subject : TSubjectAbstract read FSubject write SetSubject ;
end;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
procedure TObserverAbstract.SetSubject( const Value: TSubjectAbstract);
begin
  FSubject := Value;
  Subject.AttachObserver( self ) ;
end;
```

```
procedure TFormObserverBarGraph.DataToObserver ;
var
  i           : integer ;
  lStockTrans : TStockTrans ;
begin
  chart1.series[0].clear ;
  with (Subject as TPortfolio) do begin
    for i := 0 to Stocks.Count - 1 do begin
      lStockTrans := Stocks.Items[ i ] ;
      chart1.series[0].add(lStockTrans.Value,lStockTrans.StockCode,GetColour(i));
    end ;
  end ;
end;
```

➤ *Listing 7*

If you have loaded the source code from the companion disk, you will see that the application performs well with a single instance of each observer on the screen. All the observers dynamically change as a scroll bar is adjusted with the mouse. Every time we make even a small change to the subject, all the observers must update themselves.

What would happen if the subject was very complex, or the observers took more processing power to repaint themselves? Try opening ten instances of each observer, click `Windows | Tile`, and experiment with moving the scrollbars. The performance of the system is ground down by all the unnecessary updating of observers. There are several solutions to this problem.

The simplest solution is to add a `TTimer` to the edit form so that each time a value is changed the timer is restarted. The timer may wait for a second before updating the subject and sending the message to update the observers. This gives the user the feeling that all the observers change with each click of the mouse, but unnecessary updates to all observers while the user is changing a value are avoided.

The more sophisticated solution is for the subject to know exactly what changed and notify the observers of the details of the change, instead of forcing them to re-read all the subject's data and repaint themselves totally from scratch.

The timer solution is very easy to implement and is my preferred option in most circumstances.

The idea of notifying the observers of what has changed may be elegant, but requires careful design work and plenty of effort to implement. If the subject is to notify the observers of what has changed, it is necessary for the subject to have some knowledge of what the observer is displaying. This will lead to tighter coupling of the subject to the observer and should be avoided.

More intelligence must be built into our framework to improve the efficiency of updates. There are three places where we can put this intelligence.

First, it can be put in the code that is updating the subject. The timer approach suggested above is a quick and easy solution.

Second, the observer could remember the state of the subject

when it last checked and compare old and new values, only repainting the values that have changed. This is easy to do with a listview's `OnData` event (as long as we are changing values, not adding new ones or deleting existing ones), but much harder to do with the `TChart` based observers.

Third, the subject could have a `SetValue` method attached to each property, which would be responsible for sending the appropriate message to the observers. This would work for the share portfolio example, but would be challenging to implement once we expand the application to allow the creation or deletion of rows, or the nesting of objects within each other.

Another complication that is worth considering is how to handle the situation where the subject is contained in another application, such as a COM server, or a remote multi-tier application server. The success of our implementation depends on the subject being able to maintain a list of pointers to the interested objects. This becomes more difficult within a distributed environment.

Push And Pull Observers

In the implementation of the observer we have discussed, the subject notifies all its observers that a change has taken place and it is up to the observers to interrogate the subject to work out what has changed, and how to repaint themselves. This is known as a 'pull' implementation because the observer must 'pull' the necessary information out of the subject.

The 'pull' observer is easy to implement, but may be inefficient because of the processing overhead it causes. Every observer must determine what part of the subject has changed, or alternatively not bother to work out what has changed, and repaint itself from scratch with each and every notification.

A more efficient implementation of the observer, often called a 'push' implementation for obvious reasons, requires the subject to send details of what has changed to the observers. This will reduce the chance of unnecessary updates, but causes tighter coupling between the subject and observers. The 'push' observer is also much more challenging to implement than the 'pull' observer.

Summary

We have studied a simple implementation of a 'pull' observer that is both easy to implement and encourages loose coupling between the subject and the observers.

We have seen that while the pull observer has several advantages, there are situations where it may be too inefficient because of the need for observers to unnecessarily repaint themselves in response to minor changes in the subject. We have looked at several solutions to this inefficiency, including the use of a timer to buffer the updates as well as the use of a push observer, which is the most efficient, but is much more difficult to implement.

```
procedure TFormObserverGrid.DataToObserver;
var
  i           : integer ;
  lListItem   : TListItem ;
  lStockTrans : TStockTrans ;
begin
  LV.Items.Clear ;
  with ( Subject as TPortfolio ) do begin
    for i := 0 to Stocks.Count - 1 do begin
      lStockTrans := Stocks.Items[ i ] ;
      lListItem := LV.Items.Add ;
      lListItem.Caption := lStockTrans.StockCode ;
      lListItem.SubItems.Add( lStockTrans.StockName ) ;
      // Add more subItems . . .
    end ;
  end ;
end;
```

➤ *Above: Listing 8*

➤ *Below: Listing 9*

```
procedure TFormObserverGrid.DataToObserver;
var
  i           : integer ;
  lListItem   : TListItem ;
  lStockTrans : TStockTrans ;
begin
  LV.Items.Clear ;
  with ( Subject as TPortfolio ) do begin
    for i := 0 to Stocks.Count - 1 do begin
      lStockTrans := Stocks.Items[ i ] ;
      lListItem := LV.Items.Add ;
      lListItem.Caption := lStockTrans.StockCode ;
      lListItem.SubItems.Add( lStockTrans.StockName ) ;
      // Add more subItems . . .
    end ;
  end ;
end;
```

Good luck with the observer pattern. Please do contact me if you have any comments, or would like to discuss this pattern in more detail.

Acknowlegements

The content of this article was inspired by my 'pattern mentor', Darius Zakrzewski who hosts the Melbourne Patterns Group, a discussion group that meets twice a month to discuss the evolution of the pattern language of programming (PLOP) and the use of design patterns to solve real-life business problems. The Melbourne Patterns Group website is at www.win32dev.com/patterns/. It contains some useful links to other sites that discuss the use of design patterns in software development.

Peter Hinrichsen is a Certified Inprise Consultant and director of TechInsite P/L, a software company that specialises in client/server and objected oriented application development in Delphi. Peter works out of Melbourne, Australia and may be reached at peter_hinrichsen@techinsite.com.au

Design Patterns – Element of Reusable Object Oriented Software, by Gamma, Helm, Johnson & Vlissides. Four authors, hence the name 'Gang of Four' or GoF.