The memento pattern in practice

Michaël Van Canneyt

February 2, 2009

Abstract

People used to write database applications will sorely miss the cancel feature which is present in TDataset when they switch to object-oriented programming and use of the mediating views pattern. In this article, a solution to this problem is presented.

1 Introduction

In previous issues of Toolbox (**Joerg, a reference to the first article, and to the article about observers and MGM by Graeme**) the observer pattern and the corresponding Mediating Views pattern were presented: it was shown how the mediating views allow to connect visual controls (such as edit controls) to various properties of an object (such as the text in a label on a cover): the objects in the mediating view pattern propagate any changes done by the user directly to the object and property the control is linked to.

In contrast, in a usual database application, one can enter data, and only when the data is saved (to the local data buffer or the database itself), are the changes that were made, committed irrevocably. This means that while the data has not been saved, the changes can be undone any time by reverting to the record as it was before the editing operation was started.

The observer pattern and the mediating views are constructed in such a way that changes made in the editing controls are immediatly propagated to the object which is being edited: there is no memory of the properties of the object as they were before the editing started. This means that the changes cannot be undone. Some exceptions to this rule may exist, for instance when a previous version of the object was stored in a database: the previous version can then be reloaded from the database. Of course, by doing so, any previous changes that were made but not yet saved to the database are also undone.

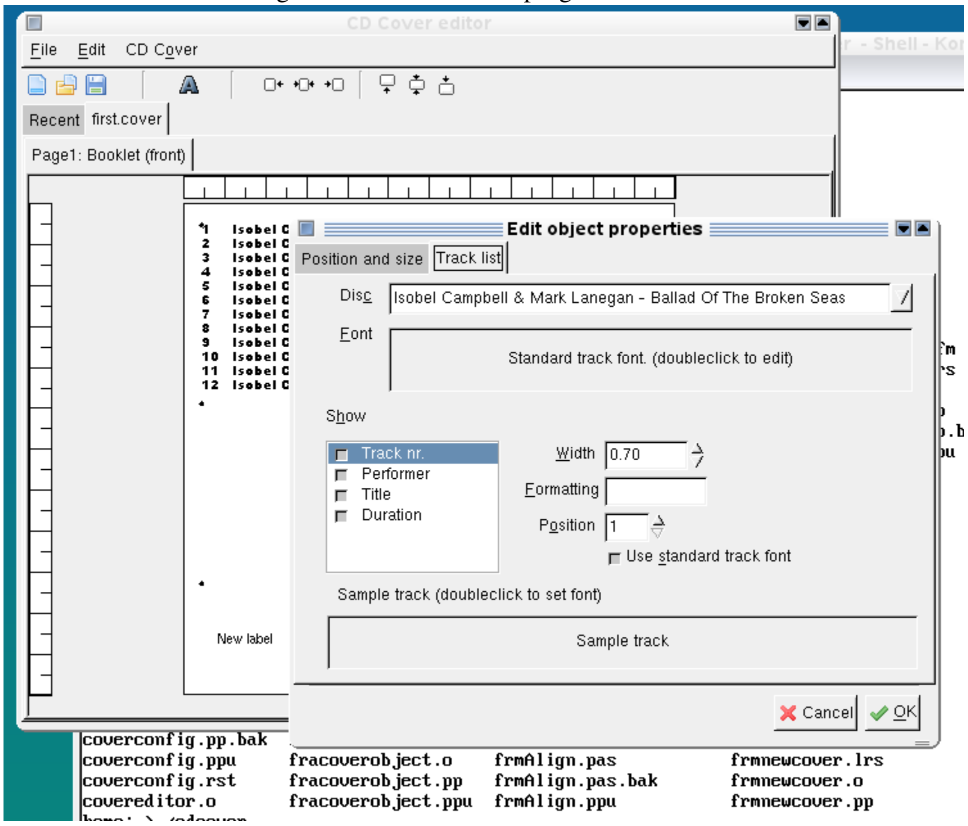
In a graphical application such as the previously presented CD-Cover application or indeed most drawing programs, even this last possibility does not exist. The CD-Cover designing program will therefor be used to demonstrate a technique to add a 'cancel' functionality to dialogs such as seen in figure 1 on page 2, when programming with the mediating view pattern.

2 The Memento pattern

To remedy the lack of cancelling, the Memento pattern can be applied: prior to editing an object in a dialog, a snapshot of the objects' properties is made. If the dialog is cancelled, the object is asked to restore its properties from the snaphot.

An alternative to the memento pattern is creating a duplicate of the object: for example, the tiOPF framework uses this approach: it is called 'Cloning' an object. While the 2 patterns

Figure 1: The CD-Cover program in action



Disc: Isobel Campbell & Mark Lanegan - Ballad Of The Broken Seas

Width: 0.70

Position: 1

```
coverconfig.pp.bak
coverconfig.ppu     fracoverobject.o    frmAlign.pas       frmnewcover.lrs
coverconfig.rst     fracoverobject.pp   frmAlign.pas.bak   frmnewcover.o
covereditor.o       fracoverobject.ppu  frmAlign.ppu       frmnewcover.pp
```

resemble each other in philosophy, the main difference between these two approaches is that the clone should be an exact replica of the object that is being edited: the clone is of the same class as the original, with exact the same properties and values for these properties, even with the same dependendent classes. In contrast, in the memento pattern a new memento class instance is created: it does not have to be the same class as the original object at all. Obviously, most properties in the original class will be duplicated in the memento class. The parent of all memento classes could look like this:

```
TMemento = Class(TObject)
  Constructor CreateMemento; virtual;
end;
TMementoClass = Class of TMemento;
```

The above `TMemento` class will be the base class for all mementos presented below.

Now, to integrate the memento pattern in the CD-Cover application, all components in the CD-Cover application's data classes are made a descendent not of `TObservedComponent`, as previously required by the use of the observer pattern, but instead they will decend of a new class, which is inserted below `TObservedComponent` and implements the basis of the memento pattern. The class is called `TCoverComponent`:

```
TCoverComponent = Class(TObservedComponent)
Protected
  Class Function GetMementoClass : TMementoClass; virtual;
  Procedure DoApplyMemento(AMemento : TMemento); virtual;
  Procedure DoFillMemento(AMemento : TMemento); virtual;
Public
  Function GetMemento : TMemento;
  Procedure ApplyMemento(AMemento : TMemento);
end;
```

This approach this makes sure that any dialog which needs to have cancel functionality added, only needs to know only the `TCoverComponent` class, no matter what the specific object instance is which is edited by the dialog.

The `TCoverComponent` has 2 public methods, the first of which is `GetMemento`: this function returns an instance of the `TMemento` class, filled with the properties of the `TCoverComponent` descendent, as they are at the moment of the call.

The second function is `ApplyMemento`. This method accepts as an argument an instance of `TMemento` - which should have been created by a call to `GetMemento` - and restores the properties with the values found in the `TMemento` instance.

The 3 virtual methods are there to make sure that each descendent creates and fills the appropriate `TMemento` descendent instance. The `GetMemento` implementation makes this more clear:

```
function TCoverComponent.GetMemento: TMemento;
begin
  Result:=GetMementoClass.CreateMemento;
  DoFillMemento(Result);
end;
```

The first line uses the class function `GetMementoClass` to create an instance of a `TMemento` descendent. For this to work correctly, the virtual `CreateMemento` constructor is used. Descendent classes of `TMemento` can use the constructor to create additional classes - this will be demonstrated later.

After the instance is created, the state of the component is saved in the `DoFillMemento` method. This method must be implemented by descendent objects, and should fill the properties of the memento instance.

The `ApplyMemento` procedure checks the validity of the `TMemento` instance passed to it, and then calls `DoApplyMemento` to actually apply the memento:

```
procedure TCoverComponent.ApplyMemento(AMemento: TMemento);
begin
  If Not (AMemento is GetMementoClass) then
    Raise Exception.CreateFmt(SErrMementoNotApplicable,
                               [AMemento.ClassName,ClassName]);
  DoApplyMemento(AMemento);
  Changed;
end;
```

After the memento was applied, the 'Changed' method is used to notify observers that the object has changed. If the memento is of the wrong class, an exception is raised, with an informative message.

3 Memento Usage in a GUI

From the GUI point of view, everything is now ready to use the memento pattern. To demonstrate this, the dialog for editing the properties of an object dropped on one of the CD-cover pages, is modified. The declaration of the object editing dialog in the CD-Cover designing applications is as follows:

```
TObjectEditorForm = class(TForm)
  procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);
  procedure FormDestroy(Sender: TObject);
private
  FCoverObject: TCoverObject;
  FMemento : TMemento;
  procedure SetCoverObject(const AValue: TCoverObject);
  { private declarations }
public
  { public declarations }
  Property CoverObject : TCoverObject Read FCoverObject Write SetCoverObject;
end;
```

This is not very different from the original declaration of the form. In fact, only a `OnClose` event handler and a private field `FMemento` are added.

The `SetCoverObject` method is slightly modified, so that when the CoverObject is passed to the dialog, a memento is created:

```
procedure TObjectEditorForm.SetCoverObject(const AValue: TCoverObject);

begin
  if FCoverObject=AValue then exit;
  FCoverObject:=AValue;
  FMemento:=FCoverObject.GetMemento;
  // ...
end;
```

(Some irrelevant parts of the code were stripped). As can be seen, one call is made to retrieve the memento of the object that will be edited in the dialog, and this `TMemento` instance is saved in the `FMemento` field.

The `OnClose` handler of the form is used to check whether the user has cancelled the changes, in which case the memento is applied again to the object that was being edited.

```
procedure TObjectEditorForm.FormClose(Sender: TObject;
  var CloseAction: TCloseAction);
begin
  If (ModalResult<>mrOK) and Assigned(FCoverObject) then
    FCoverObject.ApplyMemento(FMemento);
end;
```

If the modal result of the dialog is not `mrOK`, then the memento is applied.

The reason the `OnClose` handler is used rather than the `OnCLick` handler of the cancel button is simple: the user can close the dialog not only through the 'Ok' and 'Cancel' buttons, but also can close the window with the button provided by Windows or the X-Windows window manager. Whatever method is used, the OnClose handler will be executed.

That's all that needs to be done in the GUI to use the memento pattern. The other dialogs in the CD-Cover applications can be changed in exactly the same way: In fact, one could use visual inheritance to create a parent form that takes care of preparing and applying the memento, while the descendent forms take care of the visual details of the form.

4 Memento usage in the data model

Now, of course all objects that make up the CD-Cover data model must be modified so they fill the memento classes with the appropriate data. For the base object `TCoverObject`, the 3 methods presented above will looks as follows when they are implemented:

```
class function TCoverObject.GetMementoClass: TMementoClass;
begin
  Result:=TCoverMemento;
end;

procedure TCoverObject.DoApplyMemento(AMemento: TMemento);
begin
  FDims:=TCoverMemento(AMemento).FDims;
end;

procedure TCoverObject.DoFillMemento(AMemento: TMemento);
begin
  TCoverMemento(AMemento).FDims:=FDims;
end;
```

Where the `TCoverMemento` class is declared as:

```
TCoverMemento = Class(TMemento)
Protected
  FDims : Array[1..4] of Double;
end;
```

A less trivial class is the memento for `TImageObject`, called `TImageMemento`:

```
TImageMemento = Class(TCoverMemento)
Protected
  FCentered: Boolean;
  FPicture: TPicture;
  FStretched: Boolean;
Public
  Constructor CreateMemento; override;
  Destructor Destroy; Override;
end;
```

Beside the fields needed to keep the data for `TimageObject`, it also needs a constructor and destructor, so it can create and destroy a `TPicture` instance:

```
constructor TImageMemento.CreateMemento;
begin
  inherited CreateMemento;
  FPicture:=TPicture.Create;
end;

destructor TImageMemento.Destroy;
begin
  FreeAndNil(FPicture);
  inherited Destroy;
end;
```

The `FPicture` instance is used as follows in the `TImageObject` class of the CD-Cover designing application:

```
procedure TImageObject.DoFillMemento(AMemento: TMemento);

Var
  M : TImageMemento;

begin
  inherited DoFillMemento(AMemento);
  M:=TImageMemento(AMemento);
  M.FPicture.Assign(FPicture);
  M.FCentered  := FCentered;
  M.FStretched := FStretched;
end;
```

As can be seen, the picture is assigned; simply copying the `FPicture` instance pointer would not save the current of `TImageObject` state at all. There are of course instances where copying a pointer is sufficient. The interested reader can look at the `TTacklistMemento` for an example.

5 Conclusion

The Memento pattern explained in this article are a necessity for any program that makes use of mediating views: the CD-Cover application is no exception to this. The memento pattern makes sure the 'Cancel' buttons perform as expected. It is possible to imagine variations of the pattern: the approach taken here, with 3 methods implemented in the data class is certainly not the only one: all properties can be copied to the memento class in it's constructor, or one could create an automated routine which copies any simple published properties it finds in both classes.

The pattern has more uses: it is not hard to see that the pattern can also be used to create an undo stack. Something which could be expanded on in another contribution.