The undo stack and reusing the memento pattern

Michaël Van Canneyt

March 31, 2009

Abstract

In a previous contribution, the memento pattern was introduced. In this article, the memento pattern is used in an undo stack implementation. The showcase application is once more the CD-Cover designer application.

1 Introduction

When working with the mouse - or generally when working with a computer, an error is quickly made. At such times, an ability to undo the previous operations is a welcome addition to the application. In a previous article treating mediators and the memento pattern it was shown how to cancel operations that were performed in a dialog box using the memento pattern. In essence, the memento pattern tells us that each object can make a lightweight copy of itself, enough to restore the previous state when changes are made.

In this article, the ability to cancel operations will be taken a step further, and a simple undo stack is implemented. Rudely speaking, the undo stack records ever action taken in the application, and keeps enough information to restore the state of the cd-cover as it was prior to the start of the action.

It is easy to see that it should suffice to create a memento of all the objects that are manipulated, and keep them in a long list. As soon as an undo is requested, the memento is again applied to the objects, restoring them to their previous state.

However, while correct, such a scheme would use up a lot of resources: a lot of mementos would be kept in memory. So a more refined implementation is needed, in which the memento is of course used when appropriate.

2 Identifying the actions

It is important to know which actions must be recorded. Fortunately, in the CD-Cover designing application, there are not so much actions, basically they are the following:

1. Adding an object.
2. Editing an object's properties.
3. Resizing one object.
4. Moving selected objects. This actually includes an align operation.
5. Deleting selected objects.
6. Resizing selected objects.

The reverse operations are actually equally simple:

1. Deleting the added object.
2. Restoring the properties as they were prior to the property dialog. Here the memento pattern comes in.
3. Restoring the previous size and position of an object.
4. Restoring the previous position of the selected objects.
5. Re-creating the deleted objects. This can be achieved by not actually destroying the objects in memory, but simply by removing them from the CD-Cover but keeping them in memory.
6. Restoring the previous sizes and positions of the selected objects.

As can be seen, many operations involve restoring the previous position and size of an object. Creating a memento for such operations would be quite a waste of resources.

3 The basic undo stack objects

The undo stack basically consist of 2 objects:

1. A class representing the action to be undone. We'll name this class `TCDCoverAction`
2. A list object which keeps a list of `TCDCoverAction` instances. It should be possible to set a maximum number of items in the list. Basically 2 actions can be performed on the list: add an object (making sure the total number of items doesn't exceed the maximum), and undoing the last added action: it works according to the LIFO principle. The list class will be called `TCDCoverActions`.

It is tempting to implement these classes using a Collection class, but since the various actions will be different descencents of `TCDCoverAction`, this is not really a good idea.

Instead, both classes will be simply descendents of `TObject`. The `TCDCoverActions` class will use a `TFPList` class to maintain the list.

The `TCDCoverAction` needs only 2 methods:

```
TCDCoverAction = Class(TObject)
Public
  Procedure Undo; virtual; abstract;
  Function DisplayName : String; virtual abstract;
  Function Description : String; virtual;
end;
```

The `Undo` procedure should perform the undo action. It must be overridden by descendent classes. The `DisplayName` function should return a description of what the undo class does - in general, for example 'Delete object'. The `Description` function can return more information: it can for instance give the type of the object that the action acts on, for example 'Delete Tracklist'. By default it simply calls `DisplayName`.

The two last functions can be used in menu item captions and button hints, to give more information than the general 'Undo'.

The `TCDCoverActions` class is slightly more complicated:

```
TCDCoverActions = Class(TObject)
Public
  Constructor Create(AMaxCount : Integer);
  Destructor Destroy; override;
  Function UndoLastAction : Boolean;
  Function LastAction : TCDCoverAction;
  Function AddAction(AAction : TCDCoverAction) : Integer;
  Property Actions[Index : Integer] : TCDCoverAction;
  Property ActionCount : Integer;
  Property MaxCount : Integer;
end;
```

The meaning of these methods and properties should be quite clear from their names. Note that the maximum number of items in the undo stack is passed on to the constructor. The `UnfoLastAction` will return `True` if the last added action was succesfully undone. The `Actions` array property gives indexed access to all the actions on the stack - it's a zero based index, with `ActionCount-1` as the index of the last element - the last element can be immediatly retrieved with the `LastAction` function. This is the action that will be undone when `UndoLastAction` is called.

4 2 general descendents

When looking at the actions that should be undone, one notices that there are 3 actions which act on a single CD-Cover object, and 3 actions which act on a list of selected objects. Therefor it makes sense to create 2 abstract descendents of `TCDCoverAction`: one which keeps a pointer to a CD-Cover object, one which keeps a list of CD-Cover objects (the selection at the time of the action).

These classes are defined as follows:

```
TSingleObjectAction = Class(TCDCoverAction)
Public
  Constructor Create(AObject : TCoverObject); virtual;
  Property CoverObject : TCoverObject;
end;
```

The constructor simply keeps a reference to the `AObject` instance, which can be accessed through the `CoverObject` property.

The second class is similarly simple:

```
TSelectedObjectsAction = Class(TCDCoverAction)
Public
  Constructor Create(ASelection : TSelectionlist); virtual;
  Destructor Destroy; override;
  Property Objects : TSelectionList;
end;
```

The constructor in this case makes a copy of the selectionlist passed to it. It will clear the copy when it is destroyed, but will not free the objects in the list.

5 Actual undo actions

With these 2 objects in place, the rest of the undo classes are actually quite simple. The simplest one is the 'Add object' class, `TAddObjectAction`:

```
TAddObjectAction = Class(TSingleObjectAction)
  Procedure Undo; override;
  Function DisplayName : String; override;
end;

procedure TAddObjectAction.Undo;
begin
  FreeAndNil(FObject);
end;

function TAddObjectAction.DisplayName: String;
begin
  Result:='Add object';
end;
```

The `Undo` procedure simply frees the added object. The object hierarchy will do the rest. The `DisplayName` function is simplicity itself. Note that when the `TAddObjectAction` instance is freed, the added object itself is not freed.

When editing the property of a CD-Cover object, a `TEditObjectAction` is created:

```
TEditObjectAction = Class(TSingleObjectAction)
private
  FMemento : TMemento;
Public
  Constructor Create(AObject : TCoverObject); override;
  Destructor Destroy; override;
  Procedure Undo; override;
  Function DisplayName : String; override;
end;
```

All the interesting things happen in 2 the constructor and the `Undo` procedure:

```
 constructor TEditObjectAction.Create(AObject: TCoverObject);
begin
  inherited Create(AObject);
  FMemento:=AObject.GetMemento;
end;

procedure TEditObjectAction.Undo;
begin
  CoverObject.ApplyMemento(FMemento);
end;
```

The constructor creates a memento instance from the object that was passed to it, and saves the memento for future use: The `Undo` procedure uses this memento to restore the properties of the object as they were prior to the edit action. Finally, the destructor removes the memento from memory:

```
destructor TEditObjectAction.Destroy;
```

```
begin
  FreeAndNil(FMemento);
  inherited Destroy;
end;
```

The `TResizeObjectAction` is similar in working to the `TEditObjectAction` class and will not be treated here. Instead, the `TDeleteObjectsAction` class implementation will be shown:

```
TDeleteObjectsAction = Class(TSelectedObjectsAction)
  Constructor Create(ASelection : TSelectionList); override;
  Destructor Destroy; override;
  Procedure Undo; override;
end;
```

The constructor keeps a copy of the parent of the objects: this is the actual cover page on which the objects are positioned.

```
constructor TDeleteObjectsAction.Create(ASelection: TSelectionList);

begin
  inherited Create(ASelection);
  If (ASelection.Count>0) then
    FParent:=ASelection[0].GetParentComponent as TCoverPage;
end;
```

The 'Delete' operation in the cover editor is modified. It will no longer simply free the objects, but instead will simply remove them (using `RemoveComponent`) from the object that owns them, the `TCDCover` class. Keeping this in mind, The `Undo` procedure is quite simple:

```
procedure TDeleteObjectsAction.Undo;

Var
  I : Integer;
  O : TCoverObject;
  C : TCDCOver;

begin
  If Assigned(FParent) then
    begin
    C:=FParent.Owner as TCDCover;
    For I:=Objects.Count-1 downto 0 do
      begin
      O:=Objects.CoverObjects[I];
      C.InsertComponent(O);
      O.SetParentComponent(FParent);
      Objects.Delete(i);
      end;
    end;
end;
```

All the objects in the selection are simply re-inserted in their former owner class, and their parent component is set, using the copy that was stored in the constructor. Since the CDCover is also the `Owner` of the cover pages, it can be retrieved using the cover pages'

`Owner` property. After each object is inserted in the tree, it is removed from the list of objects. This is important, because the destructor will free any objects left in the list:

```
Destructor TDeleteObjectsAction.Destroy;

Var
  I : Integer;

begin
  For I:=0 to Objects.Count-1 do
    Objects[i].Free;
  inherited Destroy;
end;
```

This ensures that if the action is not undone, the objects (which are no longer owned by the CD-Cover instance) are freed from memory when the action is freed from memory.

The `TMoveObjectsAction` should restore the positions of the moved objects in the selection. It does this by storing a copy of the position of all objects in the selection:

```
TObjectOrigin = Record
  X,Y : Double;
end;
PObjectOrigin = ^TObjectOrigin;

TMoveObjectsAction = Class(TSelectedObjectsAction)
private
  FOrigins : PObjectOrigin;
Public
  Constructor Create(ASelection : TSelectionList); override;
  Destructor Destroy; override;
  Procedure Undo; override;
end;
```

When the action is created, it allocates memory for all the positions, and copies the positions:

```
constructor TMoveObjectsAction.Create
     (ASelection:  TSelectionList);

Var I : Integer;

begin
  inherited Create(ASelection);
  FOrigins:=GetMem(ASelection.Count*SizeOf(TObjectOrigin));
  For I:=0 to ASelection.Count-1 do
    begin
    FOrigins[I].X:=ASelection[i].Left;
    FOrigins[I].Y:=ASelection[i].Top;
    end;
end;
```

In the `Undo` operation, the positions are then simply copied back:

```
procedure TMoveObjectsAction.Undo;
```

```
Var
  I : Integer;

begin
  For I:=0 to Objects.Count-1 do
    begin
    Objects[i].Left:=FOrigins[I].X;
    Objects[i].Top:=FOrigins[I].Y;
    end;
end;
```

Obviously, the destructor should free the memory allocated in the constructor. The other undo classes act in a similar manner, and will not be discussed in detail here.

6 Using the undo stack class

The undo stack class `TCDCoverActions` has actually only 1 interesting method, and this is the `UndoLastAction` procedure:

```
function TCDCoverActions.UndoLastAction: Boolean;

Var
  A : TCDCoverAction;
  I : Integer;
begin
  A:=LastAction;
  Result:=A<>Nil;
  If not Result then
    Exit;
  try
    A.Undo;
  finally
    FActions.Remove(A);
  end;
end;
```

It looks up the last action in the list, and calls it's undo method. Then it removes the action from the stack. Since the internal `FActions` list (a `TObjectList` instance) frees the action when it is removed from the list, the action is freed after it was undone. This is also used to ensure that no more than `MaxCount` elements are on the stack;

```
function TCDCoverActions.AddAction(AAction: TCDCoverAction): Integer;
begin
  If ActionCount=MaxCount Then
    FActions.Delete(0);
  FActions.Add(AAction);
end;
```

If maxcount is reached, the first added action is removed from the list (and hence freed) when a new action is added to the list.

When using this object, a choice must be made. The CD-Cover application allows to design more than one CD-Cover at once. There are therefor 2 options:

- Keep a single global undo stack
- Keep an undo stack per cover.

The latter is chosen, as it gives more flexibility.

So, the `TCoverEditor` class (introduced in a previous article) is enhanced with a `TCDCoverActions` instance (it is created in the constructor, and freed in the destructor), and gets 2 public methods:

```
Function CurrentUndo : String;
Function UndoLastAction : String;
```

With implementation:

```
function TCoverEditor.CurrentUndo: String;

Var
  A : TCDCoverAction;

begin
  A:=Factions.LastAction;
  If Assigned(A) then
    Result:=A.Description;
end;
```

`FActions` is the `TCDCoverActions` instance maintained by the cover editor instance. The above function can be used to update the caption of the 'Undo' menu item.

```
function TCoverEditor.UndoLastAction: String;

Var
  A : TCDCoverAction;

begin
  A:=Factions.LastAction;
  If Assigned(A) then
    begin
    Result:=A.Description;
    If not Factions.UndoLastAction then
      Result:='';
    end;
end;
```

This function undoes the last action, and returns the description of the action that was undone. This method is called by the 'Undo' menu item.

These methods can be used to undo actions that are on the 'Undo' stack, all that is left is to fill the undo stack. For this, the various methods in the `TCoverEditor` class must be modified. A simple one is the `EditObject` method, invoked when a user doubleclicks an object on the CD-Cover:

```
function TCoverEditor.EditObject(O: TCoverObject): Boolean;

Var
  A : TEditObjectAction;
```

```
begin
  A:=TEditObjectACtion.Create(O);
  With TObjectEditorForm.Create(Application) do
    try
      CoverObject:=O;
      Result:=ShowModal=mrOK;
      If result then
        begin
        FActions.AddAction(A);
        Modify;
        end
      else
        A.Free;
    finally
      Free;
    end;
end;
```

As can be seen, a `TEDitObjectAction` is created, and passed the object that will be edited. If the user confirmed the editing dialog, the action is added to the undo action stack. If the user cancelled the editing dialog, the action is discarded (since nothing has changed).

When resizing an object, some extra checks are needed before adding the action to the stack, because the resize method is called repeatedly as the mouse is dragged:

```
procedure TCoverEditor.SizeObject(ACanvas : TCanvas; DX,DY : Integer);

Var
  CO : TCoverObject;
  A : TReSizeObjectAction;

begin
  CO:=FSelection[0];
  If (FActions.ActionCount=0)
      or (not (FActions.LastAction is TReSizeObjectAction))
      or (TReSizeObjectAction(Factions.LastAction).CoverObject<>CO) then
    begin
    A:=TReSizeObjectAction.Create(CO);
    FActions.AddAction(A);
    end;
```

(only the relevant code of the method is shown). A new resize action is only started when this is either the first action, or the last action was not a resize action on the same class. If the resize action is still not finished, no new action is started. A similar result could have been obtained by recording the initial size, and only creating the action when the user releases the mouse after the resize operation is finished.

As remarked earlier, the behaviour of the 'Delete Selection' method (when the user deletes te selection e.g. by pressing the 'Delete' key), has to be changed so the objects are no longer freed, but are simply removed from the CD-Cover and inserted into the undo stack:

```
procedure TCoverEditor.DeleteSelection;

Var
```

```
  I : Integer;
  B : Boolean;
  A : TDeleteObjectsAction;

begin
  B:=FSelection.Count>0;
  If B then
    begin
    A:=TDeleteObjectsAction.Create(FSelection);
    FActions.AddAction(A);
    For I:=0 to FSelection.Count-1 do
      begin
      FSelection[i].SetParentComponent(Nil);
      FCDCover.RemoveComponent(FSelection[i]);
      end;
    ClearSelection;
```

The rest of the method is unchanged. The constructor of `TDeleteObjectsAction` has copied all objects in an internal list, ready to be reinserted in the CD-Cover when the action is undone.

The other actions are created in other methods in similar ways: the action is created, passing to the constructor the manipulated object or the current selection list. After that, the action is added to the action list.

After all that, only a menu item in the main form is needed, which invokes the `UndoLastAction` method of `TCoverEditor` and all is done. These are standard operations which will not be discussed here. The interested reader is referred to the sources of the application, available on the disc that accompagnies this issue.

7 Conclusion

Adding an undo stack to an application may seem a daunting task, but turns out to be not so hard. The memento pattern - which was already used in the application - comes in handy, although resource usage may force the programmer to do some extra coding. Note that the simple undo stack here is not suitable for a 'redo' action, and that the actions may not be randomly undone; Adding those features would require extra coding.