Creating a CD-Cover program

Michaël Van Canneyt

December 4, 2008

Abstract

In the previous issue of toolbox, a number of techniques available to Lazarus programmers were introduced: fetching data from a CDDB server, printing and the observer pattern, with the related concept of mediators. In this article, the 3 techniques will be combined to create a cd-cover printing program.

1 Introduction

In the previous Toolbox issue, 3 topics were handled that have seemingly nothing in common:

- The observer pattern and - as a corollary, almost - mediators.
- Printing in Lazarus: how to draw on screen and print the same on the printer.
- How to perform a CDDB lookup in lazarus.

Combined, these 3 techniques can be used to create a real-world example: a CD-Cover designing application. This application needs to draw (the cover) print (the same cover), needs to edit properties (observer pattern) and should be able to print a track list for the CD, so being able to retrieve the track list from a CDDB server is also useful.

In this article the design of such a CD-Cover application using the above techniques will be discussed. At the end, a working application should be the result.

It should be obvious that a complete CD-Cover application entails a bit more than the above three points, nevertheless a working example will be constructed.

2 The observer pattern

The observer pattern exists since a long time, is actually a very basic design pattern and therefor was introduced in tiOPF. However, one does not always want to use tiOPF as the basis of an application (although there are many good reasons to do so), so an independent implementation of the observer pattern has been created for use with Free Pascal or Lazarus. With a slight modification, it could be used in Delphi as well.

The pattern has been implemented in the fpobserver unit, which will be included in the Free Pascal distribution. A copy is included in the sources for this article.

The unit contains the definition of the 2 basic interfaces needed in the Observer pattern:

```
TObservedOperation =
  (ooChanged,ooFree,ooAddItem,ooDeleteItem,ooCustom);
```

```
IFPObserved = Interface [SGUIDObserved]
  Procedure AttachObserver(AObserver : TObject);
  Procedure DetachObserver(AObserver : TObject);
  Procedure NotifyObservers(ASender : TObject;
                            AOperation : TObservedOperation);
end;

IFPObserver = Interface  [SGUIDObserver]
  Procedure ObservedChanged(ASender : TObject;
                            Operation : TObservedOperation);
end;
```

As can be seen, the pattern very much resembles the one presented in the article [1]. It has an additional parameter in the `NotifyObservers` call: the `Operation`. This can be used to distinguish between different circumstances in which observers are notified:

ooChanged The item under observation has changed.

ooFree The item under observation is about to be destroyed: this can be used to remove any references to the observed object.

ooAddItem mainly intended for lists to notify the observer that an item is added to the list.

ooDeleteItem mainly intended for lists to notify the observer that an item is being removed from the list.

ooCustom To be used by the programmer to signal specific events

The `ooFree` notification is a necessary addition: although `TComponent` has a free notification mechanism, many objects do not have this. Therefor the observer pattern must handle the freeing of the observed item with care; it does this by sending a notification to all observers.

the fpobserver unit also implements a hook object, which can be used to delegate the `IFPObserved` interface to:

```
TObservedHook = Class(TObject,IFPObserved)
  Constructor CreateSender(ASender : TObject);
  Destructor Destroy; override;
  Procedure AttachObserver(AObserver : TObject);
  Procedure DetachObserver(AObserver : TObject);
  Procedure Changed;
  Procedure AddItem(AItem : TObject);
  Procedure DeleteItem(AItem : TObject);
  Procedure CustomNotify;
  Procedure NotifyObservers(ASender : TObject;
                            AOperation : TObservedOperation);
end;
```

If one wishes to create an observed class, the `TObservedHook` class can be used to make it observed. In fact, the fpobserver unit creates several observed classes, such as `TObservedComponent`:

```
TObservedComponent = Class(TComponent,IFPObserved)
private
```

```
  FObservedHook: TObservedHook;
Protected
  Property ObservedHook : TObservedHook
           Read FObservedHook Implements IFPObserved;
Public
  Constructor Create(AOwner : TComponent); override;
  Destructor Destroy; override;
end;
```

Note the `Implements` clause in the `ObservedHook` property.

The implementation of `TObservedComponent` is very straightforward:

```
constructor TObservedComponent.Create(AOwner: TComponent);

begin
  inherited Create(AOwner);
  FObservedHook:=TObservedHook.CreateSender(Self);
end;

destructor TObservedComponent.Destroy;
begin
  FreeAndNil(FObservedHook);
  inherited Destroy;
end;
```

which is a very straightforward implementation. Note that when the hook is destroyed, it will send a `ooFree` notification to all observers.

In a similar manner, an observed descendent can be made of any base class. The fpobserver unit defines at least the following classes: `TObservedPersistent`, `TObservedCollectionItem`, `TObservedCollection` - which uses the `ooAddItem` and `ooDeleteItem` operation to signal additions and removals from the collection - `TObservedList` and `TObservedObjectList` as well as `TObservedStringList`. The latter two also signal additions and removals from the list.

While it is not called explicitly so, the `TDatasource` class is actually a kind of `TObservedHook` implementation, where the `TDataset` is the subject under observation. The various DB-Aware components are observers of the `TDataset` instance.

It would make little sense to implement a `TObserverHook`: only one call needs to be implemented, and it would be more difficult to manage the hook object and any callbacks than to implement the interface manually.

3 Mediators

Mediators make extensive use of the observer pattern: they connect the business classes from the business model to the various GUI elements. To be able to do this, the mediators must observe the business model classes: this in turn means that the business model classes must implement the `IFPObserved` interface. This can be done by making them descend from one of the classes in the fpObserver unit, or by explicitly implementing the interface.

Because of this close relationship, a base mediator class has been implemented in the fpObserver unit. It is - not surprisingly - called `TBaseMediator`, and has the following public interface:

```
TMediatingEvent = Procedure(Sender : TObject;
                            var Handled : Boolean) of object;
TBaseMediator = Class(TComponent,IFPObserver)
Public
  Procedure ObjectToView;
  Procedure ViewToObject;
  class function ViewClass: TClass; virtual;
  Property Subject : TObject;
  Property View : TObject;
Published
  Property SubjectPropertyName : String;
  Property Active : Boolean;
  Property ReadOnly : Boolean;
  Property OnViewToObject : TMediatingEvent;
  Property OnObjectToView : TMediatingEvent;
end;
```

Note that the mediator implements the `IFPObserver` interface, by which it will be notified if the object under observation (indicated by the `Subject` property) changes. The `View` property will point to the GUI element that should be updated when the observed changes, or whose data should be copied to the object.

The copying of data between the object and GUI control is done in the `ObjectToView` or `ViewToObject` calls. The `ViewClass` determines the minimum class that the GUI element should have for this mediator to be able to handle it (for instance, it could be `TCustomEdit`). The active and readonly properties speak for themselves: if `Active` is `False`, no automatic updating of GUI element or Object takes place - the update can be forced by calling `ObjectToView` or `ViewToObject` manually.

If `ReadOnly` is true, data is only copied automatically from the object to the GUI element.

For the rare cases that copying data should be done manually, the `OnViewToObject` and `OnObjectToView` events exist. The mediator then plays an important role in that it calls these events whenever data must be copied, so no logic must be implemented in the form itself.

The `TBaseMediator` class is an abstract class: it has no knowledge of the `View` component, indeed it offers no actual `View` property or a name of a property of the view component that must be used for data (e.g. `Text` in the case of an edit control).

For this, a `TComponentMediator` class is implemented:

```
TComponentMediator = Class(TBaseMediator)
Public
  Procedure ViewChangedHandler(Sender : TObject);
Published
  Property ViewComponent : TComponent;
  Property ViewPropertyName;
end;
```

This mediator can in fact handle a lot of cases without any other properties. This is why it has been set up as a `TComponent`: it can be dropped on a form, it's properties can be set and it is ready to work: all that must be done is set the `Subject` property to the object that must be observed, and set the `ViewComponent` to e.g. an `TEdit` instance, and the `ViewPropertyName` can be set to `Text`. The `ViewChangedHandler` method can be used to hook into an `OnChange` handler of the `ViewComponent` control: it will invoke the `ViewToObject` method.

Using all these properties, this mediator is able to copy the content of the `SubjectPropertyName` published property of `Subject` to the `ViewPropertyName` published property of the `ViewComponent` component, and vice versa. Because the properties are published, it can use Run-Time Type Information (RTTI) to do all the copying.

For example, the following code would set up a `TComponentMediator` instance to edit the first name of a person (represented by a `TPerson` object) in an edit control `EName`:

```
procedure TForm1.FormCreate(Sender : TObject);

begin
  FM:=TComponentMediator.Create(Self);
  FM.ViewComponent:=EName;
  FM.ViewPropertyName:='Text';
  FM.Subject:=TPerson.Create;
  FM.SubjectPropertyName:='FirstName';
  FM.Active:=True;
  EName.Onchange:=FM.ViewChangedHandler;
end;
```

Some controls require specialized behaviour, for instance if the data is required in a certain special format, or when there is no published property of the view or subject classes to/from which data can be copy from/to.

A nice example is a `TTreeNode` item: if one wishes to copy data between a property of a class and the `Text` property of the treenode item, then this is not readily possible because the `Text` property is not published and cannot be accessed by means of RTTI.

There are other examples. The lclmediators unit contains a number of `TBaseMediator` descendents that handle some of the standard LCL controls: `TControlMediator` is the base mediator for all control mediators. It introduces a property `DisableControl` : If this property is set to `True`, the control will be disabled if the `Subject` property is not set or any other relevant property is not set.

The classnames of the other implemented mediators make it clear which kind of control they mediate: `TEditMediator`, `TFileNameEditMediator`, `TDirectoryEditMediator` `TDateEditMediator`, `TCheckboxMediator`, `TComboBoxMediator`, `TLabelMediator` (read-only), `TSpinEditMediator`, `TTrackBarMediator` and `TMemoMediator`.

More mediators exist, but the reader is referred to the lclmediators unit for a full list.

4 CDDB Queries

The CDDB querying functionality demonstrated in the previous issue ([2]) can be abstracted out in an object which handles all the dirty work. This has been done in the `cddbquery` unit.

```
TCDDBQuery = Class(TComponent)
Public
  function QueryCD(const ADevice: String; ADisc : TCDDisc) : Boolean;
Published
  Property UseCache : Boolean Read FUseCache Write FUseCache;
  Property CacheDir : String Read FCacheDir Write SetCacheDir;
  Property User : String Read Fuser Write Fuser;
  Property HostURL : String Read FHostURL Write FHostURL;
  Property AppName : String Read FAppName Write FAppName;
```

```
end;
```

The properties speak more or less for themselves. The `HostURL` property contains the location of the `CDDB` server CGI application.

The `QueryCD` call will read the tracks from the audio CD in `ADevice` (a unix device name or a drive letter on windows), and will fill `ADisc` with the information obtained from the CDDB server. The `TCDDisc` class used in this call is discussed in the next section.

If a disc has multiple entries on the server, a dialog will be popped up to allow the user to select a matching entry.

All communication is done using the Synapse TCP/IP components, but this can be changed quite easily - only 1 call needs to be replaced.

5 The CD Cover data model

The CD Cover program is conceived around the idea of a cover project: This consists of a set of CD discs (some boxes contain more than one CD) and a set of cover pages: a booklet page, a CD box inlay page, or a CD label page.

Each CD Disc has of course a set of tracks, just as in the CDDB query data model. On each page of the CD cover, a number of printable elements can be dropped: A label (simple text), a track list, a picture or a drawing shape.

To model this CD cover project, the following classes will be used:

TCDCover This is the main component. It owns all other objects in the project.

TCDDisc A disc in the project. A project can contain multiple discs. It has 4 properties, much as in the CDDB article: `Title`, `Performer`, `Extra` and `Year`. These properties are published.

TCDCoverTracks a collection of `TCDCoverTrack` items, which represent the tracks in a disc. They have nearly the same properties as the Disc itself: `Title`, `Performer`, `Extra` and `Duration`.

TCoverPage This component represents a page in the Cover project.

TCoverObject This is the base class for all printable objects: it has descendents `TLabelObject`, `TTrackList`, `TImageObject` and `TShapeObject`. Each descendent has it's own properties.

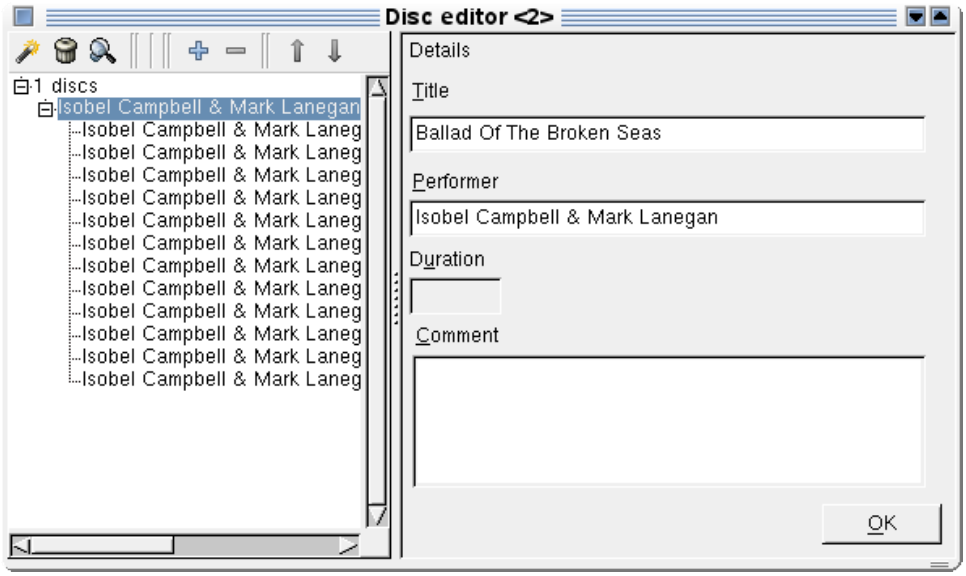
The `TCDDisc` and `TCDCoverTracks` and `TCDCoverTrack` objects contain the useful source data of the model. The last classes represent the visual objects on the printed pages. The `TCDCover` class owns all these objects, it forms the complete project.

All classes descend from `TObservedComponent`. They descend from `TComponent` because this ensures that they can be streamed: the standard Object Pascal streaming mechanism will be used to save the project to file or read it from file, just as a Lazarus or Delphi form file. They descend from `TObservedComponent` because then they can be observed, which will make it easier to write the various editing dialogs.

6 The main form

The main form will contain the menu, toolbar, and a pagecontrol with one page per opened cover project. The first page will show a list of recently opened cover files.

Figure 1: The disc editor in action



The menu will have the usual options of opening and saving a file, starting a new cover and closing one: these are all standard operations, handled using actions, but which can be found in any application and will not be discussed here.

Each page in the main form will contain a `TCoverEditor` tab sheet, which will offer an API to the main form to manage the CD Cover that is displayed on the tab sheet.

7 Managing the disc set

The `TCDCover` class has few enough methods and properties:

```
TCDCover = Class(TObservedComponent)
Public
  Property Disc[AIndex : Integer] : TCDDisc;
  Property DiscCount : Integer;
  Property Page[AIndex : Integer] : TCoverPage;
  Property PageCount : Integer;
Published
  Property Description : TStrings;
  Property Title : String;
  Property Performer : String;
end;
```

The set of discs in the project will be managed in the `TDiscEditorForm` form, which can be reached through a menu item under the "CD Cover" menu.

The `TDiscEditorForm` is displayed in action in figure 1 on page 7. The form is constructed using some actions for the buttons, a treeview to the left which shows the discs and tracks on each disc, and to the right the details of the selected item in the tree are shown.

The right-hand side of the form and the tree are managed using mediators. The right hand side is quite statical: the mediators are created once and are connected to the edit controls on the right:

```
procedure TDiscEditorForm.FormCreate(Sender: TObject);
begin
  FTitleMediator:=TEditMediator.Create(Self);
  FTitleMediator.Control:=ETitle;
  FTitleMediator.SubjectPropertyName:='Title';
  FTitleMediator.active:=True;
  FPerformerMediator:=TEditMediator.Create(Self);
  FPerformerMediator.Control:=EPerformer;
  FPerformerMediator.SubjectPropertyName:='Performer';
  FPerformerMediator.Active:=True;
  FDurationMediator:=TTimeEditMediator.Create(Self);
  FDurationMediator.Control:=EDuration;
  FDurationMediator.Active:=True;
  FDurationMediator.SubjectPropertyName:='Duration';
  FCommentMediator:=TMemoMediator.Create(Self);
  FCommentMediator.Control:=MComment;
  FCommentMediator.SubjectPropertyName:='Extra';
  FCommentMediator.Active:=True;
end;
```

Note that this could equally well be done by dropping the needed mediators on the form and connecting them to the various controls in the Object Inspector.

The tree view on the left is built with a custom routine:

```
procedure TDiscEditorForm.DisplayCover;

Var
  I,J : integer;
  D : TCDDisc;
  N : TTreeNode;

begin
  TVDiscs.Items[0].Text:=Format(SDiscCount,[FCDCover.DiscCount]);
  For I:=0 to FCDCover.DiscCount-1 do
    begin
    D:=FCDCover.Disc[i];
    N:=AddDiscToTree(D);
    For J:=0 to D.Tracks.Count-1 do
      AddTrackToTree(N,D.Tracks[J]);
    end;
  TVDiscs.Selected:=TVDiscs.Items[0];
  TVDiscs.Items[0].Expand(False);
end;
```

The `AddDiscToTree` and `AddTrackToTree` methods create the actual nodes and hook the disc or track up with the treenode using a `TTreeNodeMediator` mediator:

```
function TDiscEditorForm.AddDiscToTree(
          ADisc : TCDDisc) : TTreeNode;

Var M : TTreeNodeMediator;

begin
  Result:=TVDiscs.Items.AddChild(TVDiscs.Items[0],ADisc.Caption);
```

```
  M:=TTreeNodeMediator.Create(Self);
  M.SubjectPropertyName:='Caption';
  M.Subject:=ADisc;
  M.TreeNode:=Result;
  M.Active:=True;
end;

function TDiscEditorForm.AddTrackToTree(AParent : TTreeNode;
         ATrack : TCDCoverTrack) : TTreeNode;

Var M : TTreeNodeMediator;

begin
  Result:=TVDiscs.Items.AddChild(AParent,ATrack.Caption);
  M:=TTreeNodeMediator.Create(Self);
  M.SubjectPropertyName:='Caption';
  M.Subject:=ATrack;
  M.TreeNode:=Result;
  M.Active:=True;
end;
```

Note that the `SubjectPropertyName` property for the mediators is in both cases set to `Caption`: neither the TCDDisc nor the `TCDCoverTrack` class have this in their list of properties, so it is implemented for display purposes as a read-only published property:

```
TCDDisc = Class(TObservedComponent)
Published
  property Caption : String Read GetCaption;
end;

function TCDDisc.GetCaption: String;

begin
  Result:=FTitle;
  If (FPerformer<>'') then
    begin
    If (Result<>'') then
      Result:=' - '+Result;
    Result:=FPerformer+Result;
    end;
end;
```

And a similar implementation for the `TCDCoverTrack` class. The setup above is enough to display the contents of all discs in the treeview - at least read-only.

When an item in the tree view is clicked, the clicked item must be shown in the panel on the right-hand side of the dialog. This is done in the `OnSelectionChanged` handler of the treeview:

```
procedure TDiscEditorForm.TVDiscsSelectionChanged(Sender: TObject);

Var
  N : TTreeNode;
  M : TBaseMediator;
```

```
begin
  N:=TVDiscs.Selected;
  if (N=Nil) or (N=TVDiscs.Items[0]) then
    DisplayRoot
  else If TObject(N.Data) is TBaseMediator then
    begin
    M:=TBaseMediator(N.Data);
    If (M.Subject is TCDCoverTrack) then
      DisplayTrack(N,M.Subject as TCDCoverTrack)
    else If M.Subject is TCDDisc then
      DisplayDisc(N,M.Subject as TCDDisc);
    end;
end;
```

The `TTreeNodeMediator` stores itself in the `Data` pointer of the `TTreeNode` it is associated with. This allows to get the reference to a mediator, and hence to the mediated object: this is used in the above code to decide what kind of object was selected, and to call the appropriate display routine. As an example, the `DisplayDisc` routine looks like this:

```
procedure TDiscEditorForm.DisplayDisc(Node : TTreeNode; ADisc : TCDDisc);

begin
  FCurrentDiscNode:=Node;
  FCurrentDisc:=ADisc;
  FCurrentTrackNode:=Nil;
  FCurrentTrack:=Nil;
  FTitleMediator.Subject:=FCurrentDisc;
  FPerformerMediator.Subject:=FCurrentDisc;
  FCommentMediator.Subject:=FCurrentDisc;
  FDurationMediator.Subject:=Nil;
end;
```

A reference to the current node and disc is stored, and then the `Subject` property of the various mediators is set: The mediators will then take care of displaying the content of the various properties of the disc, or copy any changes to the disc. Note that the subject of the `FDurationMediator` mediator is set to `Nil`: because the `DisableControl` property of the mediator is set to `True`, the mediator will then disable the `EDuration` control.

The `DisplayTrack` and `DisplayRoot` routines do essentially the same as the `DisplayDisc`.

The third button to the left allows the user to scan the contents of a disc and display it on the screen at once. The code behind this button is rather simple:

```
procedure TDiscEditorForm.ScanDisc;
Var
  D : TCDDisc;
  Q : TCDDBQuery;
  N : TTreeNode;
  I : Integer;

begin
  D:=AddDisc;
  Q:=TCDDBQuery.Create(Self);
```

```
  try
    Q.UseCache:=Config.CDDBUseCache;
    Q.CacheDir:=Config.CDDBCacheDir;
    Q.User:=Config.CDDBUSer;
    Q.HostURL:=Config.CDDBHostURL;
    Q.AppName:='CDCover';
    if not Q.QueryCD(Config.CDDevice,D) then
      DeleteDisc
    else
      begin
      If (FCurrentDisc=D) then
        N:=FCurrentDiscNode
      else
        N:=FindNodeForDisc(D);
      If Assigned(N) then
        begin
        For I:=0 to D.Tracks.Count-1 do
          AddTrackToTree(N,D.Tracks[i]);
        N.Expand(False);
        end;
      end;
  finally
    Q.Free;
  end;
end;
```

It simply sets up the `TCDDBQuery` component, and calls `QueryCD`. This method does essentially what whas explained in [2], and returns the result in the `Tracks` property of a new `TCDDisc` instance (`D`). If the entry was retrieved succesfully, it is added to the tree, and the tree node for the newly added disc is expanded.

The `Config` object mentioned in the code is a `TCoverConfig` object, which is a descendent of `TObservedComponent`. This object contains the configuration of the CD-cover program in published properties. This allows the configuration dialog (found under 'File|Preferences') to use mediators to edit the configuration.

8 Drawing and adding objects

The cover can be designed in the main form: therefor, it must be able to draw itself on a canvas. Each page of the CD-Cover will draw itself on a separate canvas: the cover editor will show a pagecontrol with a page (`TTabSheet` for each page in the cover set of pages. On the page, a `TPanel` is put, and the cover will be drawn on this panel. The drawing is triggered in the `OnPaint` handler of the panel. This event handler looks like this:

```
procedure TCoverEditor.DoPaintPanel(Sender : TObject);

Var
  P : TCustomPanel;
  Page : TCOverPage;
  DPI : Integer;
  O : TPoint;

begin
```

```
  P:=TCustomPanel(Sender);
  DPI:=Screen.PixelsPerInch;
  Page:=FCurrentPage;
  O:=GetPageOrigin(P,Page,DPI);
  DrawHRuler(P.Canvas,O.X,RulerMargin,O.X+ToPoints(Page.Width,DPI),RulerWidth,DPI
  DrawVRuler(P.Canvas,RulerMargin,O.Y,RulerWidth,O.Y+ToPoints(Page.Height,DPI),DP
  Page.DrawPage(P.Canvas,O,DPI);
  DrawSelectedObjects(P.Canvas,O,DPI);
end;
```

Again, this is a quite simple routine: it uses the size of the canvas (which equals the Panel size), and uses it to center the CD Cover page. This calculation is done in the `GetPageOrigin` routine. After that, it draws a horizontal and vertical ruler on the canvas, calls `DrawPage` for the current page: It passes to this routine the canvas, the calculated origin and DPI. Finally the routine draws markers around the selected objects by calling `DrawSelectedObjects`.

The drawing of a ruler is quite simple, and the interested reader is referred to the source code that accompagnies this article. The page drawing mechanism is more interesting:

```
procedure TCoverPage.DrawPage(ACanvas: TCanvas; Origin: TPoint; ADPI: Integer);

Var
  R : TRect;
  I : Integer;

begin
  R.TopLeft:=Origin;
  R.Right:=R.Left+ToPoints(Width,ADPI);
  R.Bottom:=R.Top+ToPoints(Height,ADPI);
  ACanvas.Brush.Color:=clWhite;
  ACanvas.Brush.Style:=bsSolid;
  ACanvas.FillRect(R);
  ACanvas.Pen.Color:=clBlack;
  ACanvas.Pen.Width:=1;
  ACanvas.Rectangle(R);
  Case PageType of
    ptBookletFront : DrawBookletFront(ACanvas,R,ADPI);
    ptBookletBack  : DrawBookletBack(ACanvas,R,ADPI);
    ptInlay        : DrawInlay(ACanvas,R,ADPI);
    ptDiscLabel    : DrawDiscLabel(ACanvas,R,ADPI);
  end;
  For I:=0 to ObjectCount-1 do
    Objects[I].DrawObject(ACanvas,R,ADPI);
end;
```

It fills a white rectangle with the size of the page, and then draws a border around it. After this, it calls a page-type specific drawing routine, and finally calls `DrawObject` for all the objects that appear on this page, passing it the canvas and the rectangle in which the object should draw itself.

The page specific routine handles things such as drawing the bounding circles for the disc label:

```
procedure TCoverPage.DrawDiscLabel(ACanvas: TCanvas; PageRect : TRect; ADPI: Inte
```

```
Var
  R : Integer;
  X0,Y0 : Integer;

begin
  ACanvas.Pen.Color:=clBlack;
  ACanvas.Pen.Width:=1;
  ACanvas.Ellipse(PageRect);
  R:=ToPoints(Radius1,ADPI);
  X0:=(PageRect.Left+PageRect.Right) div 2;
  Y0:=(PageRect.Top+PageRect.Bottom) div 2;
  ACanvas.EllipseC(X0,Y0,R,R);
end;
```

The `Radius1` is a property of the page, which contains the radius of the inner boundary of the label.

The `DrawObject` routine is a virtual routine in `TCoverObject` which must be implemented by each descendent. The `TLabelObject` draws it's text using the following routine:

```
procedure TLabelObject.DrawObject(ACanvas: TCanvas; PageRect: TRect;
  ADPI: Integer);

Var
  TT : TTextStyle;
  R : TRect;
begin
  ACanvas.Font:=Self.Font;
  R.Left:=PageRect.Left+ToPoints(Left,ADPI);
  R.Top:=PageRect.Top+ToPoints(Top,ADPI);
  R.Right:=R.Left+ACanvas.TextWidth(Caption);
  R.Bottom:=R.Top+ACanvas.TextHeight(Caption);
  FillChar(TT,SizeOf(TTextStyle),0);
  TT.AlignMent:=Self.AlignMent;
  TT.Layout:=Layout;
  TT.WordBreak:=WordWrap;
  TT.Opaque:=Not Transparent;
  ACanvas.TextRect(R,R.Left,R.Top,Caption,TT);
end;
```

The whole routine is a simple preparation for the call to `TextRect`: A bounding rectangle is calculated, the `TTextStyle` record is prepared using the properties of `TLabelObject`: `Alignment`, `Layout`, `WordWrap` and `TransParent`.

Now that everything can be drawn, it is time to implement a routine to actually add an element to a page: This is done using a menu item. The 'CD Cover|Add Object' menu can be used to add an object to the page: for each type of object, there is a special menu item, constructed in the following routine :

```
procedure TMainForm.SetupObjectsMenu;

  Function AddItem(AClass : TCoverObjectClass; ACaption : String) : TObjectMenuIt
```

```
  begin
    Result:=TObjectMenuItem.Create(Self);
    Result.Caption:=ACaption;
    Result.ObjectClass:=AClass;
    Result.OnClick:=@DoObjectMenuClick;
    MIAddObject.Add(Result);
  end;

begin
  AddItem(TLabelObject,'Label');
  AddItem(TShapeObject,'Shape');
  AddItem(TTrackList,'Track list');
  AddItem(TImageObject,'Image');
end;
```

The menu item class is `TObjectMenuItem`. It can keep a reference to the correct class in the `ObjectClass` property. The `OnClick` handler just calls the current editor's `AddObject` method, and passes it the `ObjectClass` of the object that should be created:

```
procedure TMainForm.DoObjectMenuClick(Sender: TObject);
begin
  If Assigned(FCurrentEditor) and
     (Sender is TObjectMenuItem) then
   FCurrentEditor.AddObject((Sender as TObjectMenuItem).ObjectClass);
end;
```

Adding an object is a 2-step process. First, class to be added is stored and the editor is put into 'Add' mode.

```
procedure TCoverEditor.AddObject(AObjectClass: TCoverObjectClass);
begin
  FAddObjectClass:=AObjectClass;
  FEditorMode:=emAdding;
end;
```

and then the cover editor waits for a mouse click, so it knows where to add the object. In the mouse-up handler, the second step of the project is handled:

```
procedure TCoverEditor.DoPanelMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  If (Button=mbLeft) and (EditorMode=emAdding) then
    begin
    AddObjectAt(FAddObjectClass,X,Y);
    FEditorMode:=emNeutral;
    FAddObjectClass:=Nil;
    end;
end;
```

The `AddObjectAt` command is called with the class to be created and the current mouse position, and then the editor mode is reset. In the `AddObjectAt` method, the real work is done:

```
procedure TCoverEditor.AddObjectAt(AClass: TCoverObjectClass; X, Y: Integer);

Var
  O : TCoverObject;
  Ori : TPoint;
  DPI,DX,DY : Integer;
  P : TCoverPage;

begin
  DPI:=Screen.PixelsPerInch;
  O:=AClass.Create(CDCover);
  P:=FCurrentPage;
  Ori:=GetPageOrigin(FCurrentPanel,P,DPI);
  DX:=ToPoints(P.Width,DPI);
  DY:=ToPoints(P.Height,DPI);
  X:=X-Ori.X;
  Y:=Y-Ori.Y;
  if X>DX then X:=DX-ToPoints(O.Width,DPI);
  If X<0 then X:=0;
  if Y>Y+DY then Y:=DY-ToPoints(O.Height,DPI);
  if Y<0 then Y:=0;
  O.Left:=ToUnits(X,DPI);
  O.Top:=ToUnits(Y,DPI);
  O.SetParentComponent(FCurrentpage);
  ClearSelection;
  AddToSelection(O);
  FCurrentPage.DrawObject(O,FCurrentPanel.Canvas,Ori,DPI);
  DrawSelectedObjects(FCurrentPanel.Canvas,Ori,DPI);
end;
```

The object is created, and then the position of the object is calculated and corrected so it falls within the page boundaries. It is added to the page using the `SetParentComponent` method. After this, the selection is cleared and both the newly created object and the selection are drawn.

A nice example is shown in figure 2 on page 16.

9 Printing

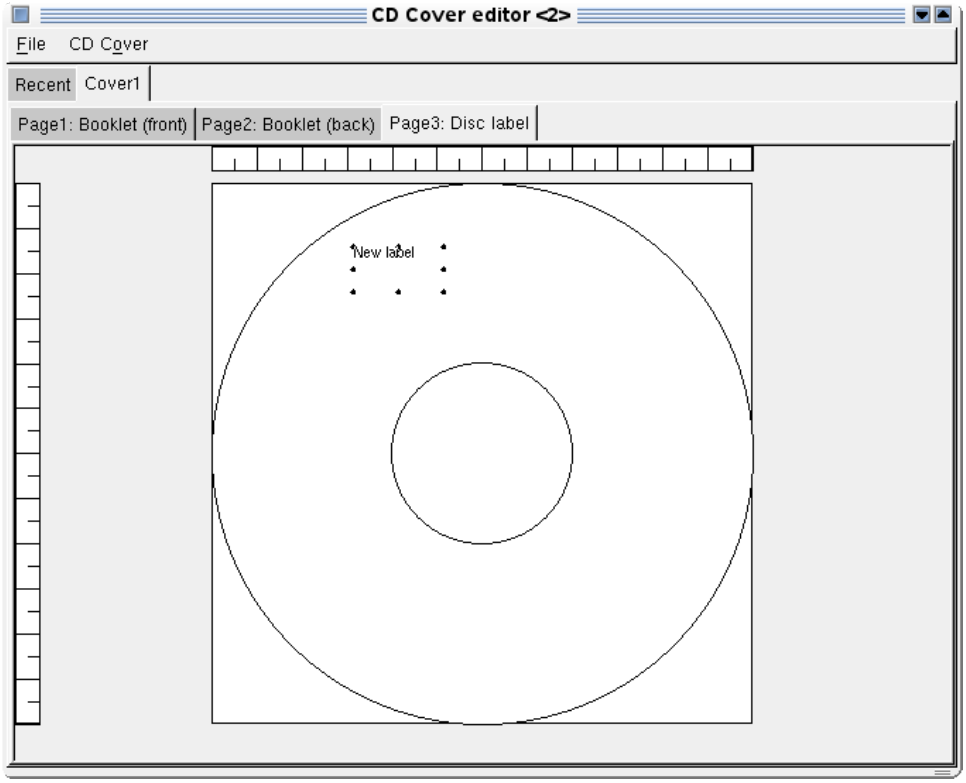
Printing the cover is actually quite simple now that the drawing routines have been implemented. To print the currently visible page, the following code can be used:

```
procedure TMainForm.PrintCurrentPage;

Var
  O : TPoint;
  P : TCoverPage;

begin
  If Not SetupPrinter then exit;
  Printer.Title := 'Cover '+FCurrentEditor.FileName;
  Printer.BeginDoc;
  try
```

Figure 2: Dropping an object on a page




```
    O.X:=ToPoints(2,Printer.XDPI);
    O.Y:=ToPoints(2,Printer.XDPI);
    P:=FCurrentEditor.Currentpage;
    P.DrawPage(Printer.Canvas,O,Printer.YDPI);
  Finally
    Printer.EndDoc;
  end;
end;
```

First, the printer setup dialog is shown. If this action was succesful, then the printer is set up, exactly as it was done in [3]. The origin is set (fixed) at a 2 centimeter margin from the top and left page borders, and the currently displayed page is printed.

10 Conclusion

While not all code has been shown here, it should be obvious that making a real-world application is quite easy using the techniques presented in the previous issues of Toolbox. Not everything has been shown: printing multiple pages (requires a page-fitting algorithm), handling multi-selection, moving of objects with the mouse. All this requires separate techniques. But adding and editing objects is quite possible: the editing of the properties of a label can be done with mediators, just as the tracks of a disc can be edited. The interested reader can review the sources on CD-rom for more details on how this can be done. The discussion of the other techniques is left for a future contribution.

References

[1] 1 Graeme Geldenhuys, The observer pattern, Toolbox 06/08

[2] 2 Michael Van Canneyt, CDDB lookups in Lazarus, Toolbox 06/08

[3] 3 Michael Van Canneyt, Printing in Lazarus, Toolbox 06/08