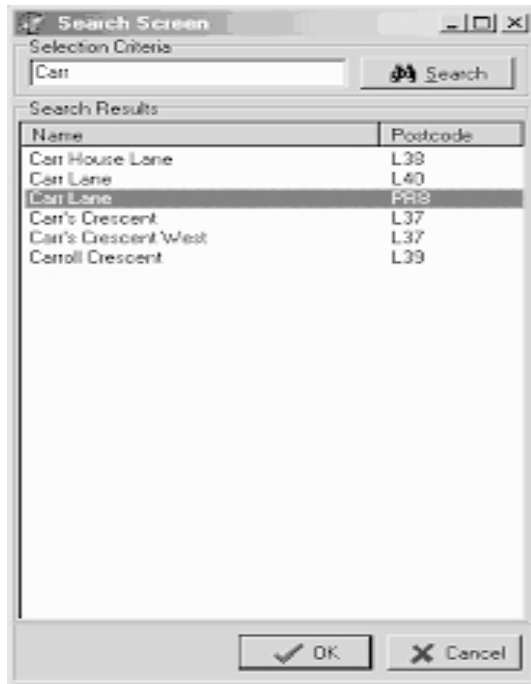


Seek and You Will Find

by Joanna Carter

Continuing this series of articles on using a Persistence Layer for managing Business Objects. We have been looking at the management of collections of objects. Having looked at the design of a List View that knows how to display a Collection of objects, we now move on to discuss the construction of a Search Dialog that allows us to find one particular object in a collection.



Here is the appearance of the Search Dialog as used in an application where there is a requirement to enter the name of a street, accurately spelled, along with the Postcode; this will then tell the program which Area the street is in.

```
TPDSearchForm = class(TForm)
  Panell: TPanel;
  btnOK: TBitBtn;
  btnCancel: TBitBtn;
  gbSearchResults: TGroupBox;
  pdlvSearch: TPDListView;
  gbSelectionCriteria: TGroupBox;
  btnSearch: TBitBtn;
  edtSearchStr: TEdit;
  procedure pdlvSearchSelectItem(
    Sender: TObject;
    Item: TListItem;
    Selected: Boolean);
  procedure pdlvSearchDbClick(
    Sender: TObject);
  procedure btnSearchClick(Sender: TObject);
  procedure btnOKClick(Sender: TObject);
  procedure FormShow(Sender: TObject);
  procedure edtSearchStrEnter(
    Sender: TObject);
```

Here is the declaration of the components and events on the search dialog; we will discuss the events in full, later on.

```
private
  fCollection: TPDCollection;
  fInternalCreate: Boolean;
  fSelectedItem: TPDObjct;
  fPropertyName: string;
  function GetSelectedItem: TPDObjct;
```

In the private section, we need to have a pointer for the Collection that we are going to be searching; another for holding a reference to the object that is selected together with an accessor function for the public property and also somewhere to hold onto the name of the property that will be used in the search criteria.

```
protected
  property Collection: TPDCollection
    read fCollection;
```

I have placed the Collection property in the protected section of the form class, so that it will not be available to anyone using the form.

```
public
  constructor Create(ACollection:
    TPDCollection;
  PropertyName: string = #0;
  SearchStr: string = #0;
  Immediate: Boolean = True); reintroduce;
  overload; virtual;
  constructor Create(ACollectionClass:
    TPDCollectionClass;
  PropertyName: string = #0;
  SearchStr: string = #0;
  Immediate: Boolean = True); reintroduce;
  overload; virtual;
  destructor Destroy; override;
  property SelectedItem: TPDObjct
    read GetSelectedItem;
end;
```

The Method Behind the...

There are two constructors that look almost identical, but provide different functionality. There may be an occasion when we want to search an existing Collection of objects and that is what the first constructor is for. On the other hand, we may want to search a Collection of objects of a particular type, not having already created the Collection; that is what the second constructor is for.

```
constructor TPDSearchForm.Create(
  ACollection: TPDCollection;
  PropertyName, SearchStr: string;
  Immediate: Boolean);
begin
  inherited Create(nil);
  fPropertyName := PropertyName;
  edtSearchStr.Text := SearchStr;
  fCollection := ACollection;
  Caption :=
    fCollection.ItemClass.DisplayName
    + ' Search Screen';
  pdlvSearch.Collection := fCollection;
  if fPropertyName <> #0 then
    fCollection.AddOrderBy(fPropertyName);
  if (SearchStr = #0) then
    fCollection.Connect
  else
    if Immediate then
      btnSearchClick(nil);
end;
```

The first thing we need to do in the constructor is to call the inherited constructor, but you will notice that we pass in nil as the Owner parameter. This means that we are going to be responsible for the memory allocation and disposal whenever we create an instance of this dialog.

The PropertyName is noted and any string that may have been in the parameter is passed to the edit box that holds the search string. If you look at the declaration of this method, you will see that I default both this and the SearchStr parameters to #0. This is to give us a means of using any default properties that are declared in the Collection for browsing as well as to help determine whether we want the search to start as soon as the dialog is shown; it also allows us to pass in ' ' (space) as a valid search string.

The Caption is set by using the DisplayName class function of TPDOject; this gives us something more user-friendly than the ClassName.

Finally, if Immediate is True, we execute the search ready for viewing as soon as the Dialog is visible.

```
constructor TPDSearchForm.Create(
    ACollectionClass: TPDCollectionClass;
    PropertyName, SearchStr: string;
    Immediate: Boolean);
begin
    fCollection := ACollectionClass.Create;
    fInternalCreate := True;
    Create(fCollection,
        PropertyName, SearchStr, Immediate);
end;
```

The constructor that takes a Collection Type rather than a pre-existing Collection is much simpler; just creating a Collection of the appropriate type and then passing that into the first constructor.

```
destructor TPDSearchForm.Destroy;
begin
    if fInternalCreate then
        begin
            fCollection.Free;
            fCollection := nil;
        end;
    inherited Destroy;
end;
```

When we look at the Destructor, we can raise the question to free or not to free? I have opted to free the collection that is held in the private field if it was created by the second constructor, otherwise we could end up freeing the Collection that was passed in before the calling code had finished with it.

Eventually...

Having taken care of the creating and destroying of the Dialog, we now need to handle any User Interaction through the event handlers on the various components.

```
procedure
    TPDSearchForm.pdlvSearchSelectItem(
        Sender: TObject; Item: TListItem;
        Selected: Boolean);
begin
    if Selected and (fCollection.Count > 0)
    then begin
        while (fCollection.Count <= Item.Index)
            and not fCollection.IsDone do
                fCollection.Next;
```

```
        fSelectedItem :=
            fCollection[Item.Index];
        end
    else
        fSelectedItem := nil;
    end;
```

When an Item is selected in the List View, we need to keep track of which Object in the Collection corresponds to that Item.

The Selected parameter tells us whether an Item has been selected or unselected and we react to that by setting fSelectedItem appropriately.

I put in the added safety of checking whether there are any Objects in the Collection before checking to see if sufficient Objects have been retrieved into the Collection to cater for the Index of the selected Item. This might seem like overkill, but rather than a run-time error; the call to fCollection.Next will add as many Objects as the 'Chunk Size' of the Collection dictates.

```
procedure TPDSearchForm.pdlvSearchDbClick(
    Sender: TObject);
begin
    if fSelectedItem = nil then
        ModalResult := mrCancel
    else
        ModalResult := mrOK;
    end;
```

If a user double-clicks on the list of objects, then we want to assume they have made a selection. As fSelectedItem will already have been set, we can then close the Dialog with the appropriate Modal Result.

```
procedure TPDSearchForm.btnSearchClick(
    Sender: TObject);
begin
    fCollection.Disconnect;
    fCollection.ClearOrderBy;
    fCollection.ClearCriteria;
    fCollection.Clear;
    fSelectedItem := nil;
```

When a user clicks on the Search button, the first thing we have to do is to clear the existing Collection completely, ready for the new Criteria that have been set and set fSelectedItem to nil.

```
if fPropertyName <> #0 then
    begin
        fCollection.AddOrderBy(fPropertyName);
        if edtSearchStr.Text <> '' then
            fCollection.AddCriteria(
                TStartingCriteria, fPropertyName,
                [edtSearchStr.Text]);
        end;
    fCollection.Connect;
```

If fPropertyName were #0, then we would not be able to know which property to search by or to specify the collection to be sorted on that property, therefore all we would do is to Connect the Collection, thereby filling the List View.

```
btnSearch.Default := fCollection.Count = 0;
btnOK.Default := fCollection.Count > 0;
```

Once the Collection is connected, we are able to detect whether there are any Objects in the Collection and set which button is to be the default button appropriately.

```
try
  pdlvSearch.SetFocus;
except
  on E: Exception do;
end;
```

Because it is possible that this code could be called before the Dialog is Visible, setting the focus can raise an exception. This method of ‘eating’ an exception is not to be recommended, but until anyone can tell me how to avoid this exception on an invisible form, I will let it stand.

```
if fCollection.Count > 0 then
  fSelectedItem := fCollection[0];
end;
```

Finally, we set fSelectedItem to the first Object in the Collection to match with the first Item in the List View being selected visually.

```
procedure TPDSearchForm.btnOKClick(
  Sender: TObject);
begin
  pdlvSearchDb1Click(Sender);
end;
```

Yes, I know I could have used Actions to write common code for closing the Dialog, but there is so little code to write, that I chose to simply redirect the Ok button to the same code as the double-click.

```
procedure TPDSearchForm.FormShow(
  Sender: TObject);
begin
  if fCollection.Count > 0 then
    pdlvSearch.SetFocus;
end;
```

When the Dialog is first shown, its default focused control will be the edit that is used for entering the search string, unless there are already Items in the List View; then the focus will be set to the List View.

```
procedure TPDSearchForm.edtSearchStrEnter(
  Sender: TObject);
begin
  btnSearch.Default := True;
  btnOK.Default := False;
end;
```

It is important to ensure that, if the user were to hit the Enter key, they would not execute an unexpected action. This code ensures that the default button is set correctly, so that the Enter key will initiate a search from the edit box rather than close the form.

Conclusion

We have designed a Search Dialog for use with Collections of Objects. We used overloaded constructors to allow us to either pass in an existing collection or search a collection of a specified type. We specified Default Parameters to enable us to do things like using ‘ ’ (space) as a valid search string. Events were handled for user interaction to allow us to set up the Dialog so that, when it was closed, we would be able to know which Object had been selected. That is after all the reason for using a Search Dialog!

Next time, I shall be looking at another Design Pattern that extends the Observer pattern. It is called the ‘MVC’ or Model/View/Controller. This very powerful pattern allows us to create multiple interfaces that are kept in sync with an underlying Object. *[Joanna is also talking about MVC at the mini-meeting at Old Trafford on September 10th. Ed]*



Joanna Carter is a writer, developer and trainer specialising in requirements-driven training and consultancy. Joanna is involved with mentoring several companies in the setting up of mapping object-oriented systems into relational databases. The Object Store

discussed in these articles is now available commercially. You can e-mail her at JoannaC@btinternet.com and her web site can be found at joannac.btinternet.co.uk