S60 application views

Introduction to S60 View Structure

View architecture is widely used in application development in Symbian. A GUI application has one or more views. The views are used to display application data and UI controls. Each view has its own control stack. Each view's container and controls are created when it is called forward, and destroyed when another view of the same application is called forward. When activating a view, a message ID and a message can be passed. This enables a great deal of potential to implement new views and utilize existing views from other applications.

View Architecture Overview

For Symbian applications there are two common view architecture solutions. The traditional architecture and the S60 view architecture. The S60 view architecture extends traditional architecture's functionality by adding view managing to base classes.

The traditional view architecture uses base class `CAknAppUi` for the UI controller. The UI controller uses a `CCoeControl` derived compound control as a view. Of course, it is possible to have multiple views in this architecture also. The traditional architecture offers more flexibility to control the views.

The S60 view architecture uses base class `CAknViewAppUi` for the UI controller. The UI controller manages one or more CAknView derived views. A view class uses a `CCoeControl` derived container control. The container class has the purpose as in the traditional view architecture. The S60 view architecture adds view management support without much extra effort. It also makes cross-application view activation possible.

Using the S60 View Architecture

In S60 view architecture the UI controller or so called AppUi class must be inherited from CAknViewAppUi. The UI controller creates views and adds them to a view stack. Also if multiple views are used, the default view must be defined.

```
void CMyAppUi::ContructL()
{
BaseConstructL(EAknEnableSkin | EAknEnableMSK);

iMyView = CMyView::NewL();
AddViewL(iMyView);
SetDefaultViewL(*iMyView);
}
```

The view class is inherited from CAknView. The view class must implement two methods; DoActivateL() and DoDeactivate(). The container class for UI controls should be created in the DoActivateL and destroyed in DoDeactivate. The command handling is usually done in the view class. The container class is responsible for creating and destroying the actual UI controls.

Two approaches in application views implementation

View recreation each time it is about to be shown

This is S60 UI application template approach. View is created before displaying and destroyed then. The drawback is obvious -- you lose the view data and you need to take some action to sync this view context with other views to make your application consistent

```
void CMyView::DoActivateL(const TVwsViewId& /*aPrevViewId*/,
      TUid /*aCustomMessageId*/,
      const TDesC8& /* aCustomMessage*/)
    {
    if (iMyContainerControl == NULL)
        {
        iMyContainerControl = CreateMyContainerL();
        iMyContainerControl->SetMopParent(this);
        AppUi()->AddToStackL(*this, iMyContainerControl);
```


(C) Copyright Nokia 2014. All rights reserved.

```
        }
    }

void CMyView::DoDeactivate()
    {
    if (iMyContainerControl != NULL)
        {
        AppUi()->RemoveFromViewStack(*this, iMyContainerControl);
        delete iMyContainerControl;
        iMyContainerControl = NULL;
        }
    }
```

1. **`DoActivateL()` and the Previous View ID.** The purpose of `DoActivateL()` in the view class is to create the container and also to handle the messages that are passed. In addition to those described in the previous section, `DoActivateL()` gets the `TVwsViewId` for the previous view. This can be used to switch back to the original view with or without a message, which may be useful within your own application views. If your application is called by another application, and when the user exits your application, the previous view is automatically restored. Knowing the previous view can be useful if you want to specify which application to respond to.
2. **`DoDeActivateL()`** is called when another view has been activated and the previous active window needs to be shut down. This order makes view switching fast. In DoDeActivateL the view is removed from the stack and therefore the view's container and its controls are destroyed.

View hiding approach - performance is better

It is quite sufficient approach when you need to represents the same data on different views. Consider situation when you have several views. First view is a dat list and the rest of the views are selected data record visualisation. Simultaneously as you browsing data list view you can update data reference in all related views. What do you need to implement that?

1. View implementation

```
void CMyView::ConstructL()
    {
 BaseConstructL(R_MYMAIN_VIEW);
 iContainer = CreateContainerL();
 iContainer->SetMopParent( this );
    }

void CMyView::DoActivateL(const TVwsViewId& aPrevViewId, TUid aCustomMessageId,
const TDesC8& aCustomMessage)
 {
 iContainer->MakeVisible(ETrue);
 AppUi()->AddToViewStackL( *this, iContainer );
 }

void CMyView::DoDeactivate()
 {
 iContainer->MakeVisible(EFalse);
 AppUi()->RemoveFromViewStack(*this, iContainer);
 }
```

2. View initialization

```
void CMyAppUi::InitializeContainersL()
 {
 CAknNavigationControlContainer* naviPane =
(CAknNavigationControlContainer*)StatusPane()->ControlL(TUid::Uid(EEikStatusPaneUidNavi));
 iTabNaviDecorator = naviPane->ResourceDecorator();
 iTabNavigator = ( CAknTabGroup* ) iTabNaviDecorator->DecoratedControl();
```


(C) Copyright Nokia 2014. All rights reserved.

```
   AddViewL(YourAppView0::NewL());
   AddViewL(YourAppView1::NewL());
   AddViewL(YourAppView2::NewL());
   YourMainView * mainView =  YourMainView::NewL();
   AddViewL(mainView);
   SetDefaultViewL(*mainView);
   iTabNavigator->SetActiveTabById(mainView->Id());
   ...
      }
```

3. View switching To switch to a view within your application, you can use the `AppUi` class method `ActivateLocalViewL()`. The parameter for `ActivateLocalViewL()` is a `TUid`. The following is an example of view switching:

```
const TUid KDemo1ViewId = { 1 }; // UID of the first view
ActivateLocalViewL(KDemo1ViewId); // activate view 1
```

A new view is activated first and the previous view is deactivated after that. This allows quick view switching to take place. When deactivating, all controls, including menus and dialogs, are also closed down. System dialogs automatically handle this. If you need to save data in your dialog when the dialog is open but the view is being shut down, you need to save the data when handling EAknSoftkeyCancel. In these cases Cancel/back route in the dialog is followed.

```
TKeyResponse CMyAppUi::HandleKeyEventL(const TKeyEvent& aKeyEvent, TEventCode aType)
 {
 TVwsViewId activeViewId;
 TInt active = iTabNavigator->TabIndexFromId(activeViewId.iViewUid.iUid);
 User::LeaveIfError(active);

 TInt count = iTabNavigator->TabCount();
 switch (aKeyEvent.iCode)
  {
  case EKeyLeftArrow:
   if (active > 0)
    {
    active--;
    ActivateLocalViewL(TUid::Uid( iTabNavigator->TabIdFromIndex( active )));
    return EKeyWasConsumed;
    }
   break;
  case EKeyRightArrow:
   if (active + 1 < count)
    {
    active++;
    ActivateLocalViewL(TUid::Uid( iTabNavigator->TabIdFromIndex( active )));
    return EKeyWasConsumed;
    }
   break;
  default:
   break;
  }
 return EKeyWasNotConsumed;
 }
```

1. Syncing with tab control

```
void CMyAppUi::HandleViewDeactivation(const TVwsViewId& aViewIdToBeDeactivated,
const TVwsViewId& aNewlyActivatedViewId)
 {
```


(C) Copyright Nokia 2014. All rights reserved.

```
 CAknViewAppUi::HandleViewDeactivation(aViewIdToBeDeactivated,
aNewlyActivatedViewId );
 iTabNavigator->SetActiveTabById(aNewlyActivatedViewId.iViewUid);
 }
```

Sending Messages

`ActivateLocalViewL()` has been overloaded to include `MessageUid` and a message. `MessageUid` is a `TUid` and it is usually used for a specific dialog page in a view or to execute a certain functionality. Messages are in `TDesC8` descriptors and can be used to pass data between the views. Here is an example:

```
const TUid KViewUid= {1};
const TUid KCustomMessageUid= {2};
TBuf8<255> customMessage;
customMessage.Copy(_L8("Some data here"));
ActivateLocalViewL(KViewUid, KCustomMessageUid, customMessage);
```

Using Other Applications' Views

To activate a view of an external application, use `ActivateViewL()` of the `AppUi` class. `ActivateViewL()` behaves in the same way as `ActivateLocalViewL()`. The only difference is that `ActivateViewL()` takes the `TVwsViewId` parameter type instead of TUid. TVwsViewId consists of the UID of the application and the UID of the view in that application. Here is an example:

```
const TUid KPhotoAlbumUid KGalleryUid = {0x101f8599 0x101F4CD1};
CCoeAppUi::ActivateViewL(TVwsViewId(KGalleryUidKPhotoAlbumUid,TUid::Uid(1)));
```

`ActivateViewL()` has been overloaded in the same way as `ActivateLocalViewL()`

To pass messages - here is an example:

```
const TUid KCustomMessageUid= {2};
TBuf8<255> customMessage;
customMessage.Copy(_L8("Some data here"));
const TUid KGalleryUid KPhotoAlbumUid={0x101f8599 0x101F4CD1};
CCoeAppUi::ActivateViewL(TVwsViewId(KGalleryUidKPhotoAlbumUid,
TUid::Uid(1)), KCustomMessageUid, customMessage);
```

When the user exits the launched application, the framework will automatically return the original view from where the `ActivateViewL()` was called.

For more details see the S60 Developer Library documentation about UI architecture: [1].

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.