

Designing and building portable UIs for Symbian OS: Using multiple controllers

Sander van der Wal, mBrain Software

Revision 1.0, March 2004

Introduction

This document is part of a series of three papers explaining how to design and write the user interface (UI) part of a Symbian OS application in such a way that it is largely portable between different Symbian OS UI platforms.

The first paper [1] described a method to design dialog interfaces that are completely independent of the UI. Dialogs are one of the major platform-dependent parts of the UI. Having a platform-independent interface means that using a dialog is no longer an obstacle in the quest to writing a platform-independent UI layer.

The second paper [2] built on the first one by describing how to extend the actual implementation of a dialog to create an even larger platform-independent UI-layer. The trick was to use a Controller (in the MVC sense). The AppUI delegated as much work as possible to the Controller. With the Controller being independent of the actual UI, it is possible to reuse it in different UI distributions of a program.

This third and final paper in the series elaborates on the Controller idea. We will look at a number of refinements that will enhance both reuse and portability of the code in a Controller.

Some remaining problems

Paper [2] introduced the idea of using a single, UI-independent Controller that took care of most of the implementation of an AppUI. As we have seen, this approach allowed us to reuse much of the code in an AppUi. Unfortunately, not all code could be reused. By analyzing the situation in Pdf+ I came to identify a number of reasons why not all of the code could be reused.

1. Some UIs need views that are not needed in other UIs. A good example of this is the file view, which is needed in the Avkon, UIQ and CKON/Series 90 UIs. This view is not necessary in the Symbian OS v5 and CKON/Series 80 UIs. As a result, you would not have put file view handling code in the Controller. On the other hand, it appears that there are reuse possibilities here. Renaming or deleting a file aren't very UI-dependent operations, after all.
2. Some UIs prescribe that you use certain built-in facilities, while others do not. Arcane as it is worded, this topic is best introduced by an example. The CKON UI has a built-in search dialog, which apps are expected to use (when they want a Nokia OK logo, that is). The way this facility must be integrated in an app prevented incorporating the Pdf+ search code into the Controller. The choice was therefore to either re-implement the CKON search dialog myself, or to find a way to reuse the existing implementation.

The solution for this problem is: use multiple Controllers for implementing the Controller functionality of a single AppView.

Using a Controller with multiple AppViews

When an application has a number of AppViews, using a single Controller prevents us from reusing as much code as possible. The example of an AppView showing files makes this clear, as some UIs demand a separate file view, while others do not. As a result, file handling code will not be a part of the Controller, because it won't be needed in a number of UI-dependent distributions of the program.

So, to make it possible to reuse more code, why not use multiple Controllers, instead of a single one? For starters, we can use a separate Controller for each AppView. This idea makes a lot of sense. Take the file view as an example. If you wanted to delete a file from this view, you would first select one, then ask the user to confirm the deletion, and finally delete the file if they confirm their intentions. Also, in a different file-based application, deleting a file is the same for all kinds of files, PDF documents, movies, MP3s. Therefore it appears that reuse across multiple programs is feasible as well.

How would this look like in code?

An AppUi's `HandleCommandL()` could look like:

```
void CPdfUIQAppUi::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
        case EPdfCmdGoToPage:
        case EPdfCmdFirstPage:
            // Other document-related commands
            iPageController->HandleCommandL(aCommand);
            break;

        case EPdfCmdDeleteFile:
        case EPdfCmdRenameFile:
            // Other file-related commands
            iFileController->HandleCommandL(aCommand);
            break;

        // Commands that are independent of the current AppView
        case EEikCmdHelpAbout:
            // What to do?
            break;

            // etc.
        default:
            break;
    }
}
```

It appears that routing commands to the appropriate Controllers is easy for commands that are related to a given AppView – you just route them to the appropriate Controller. Commands affecting the application itself, such as the 'About' dialog box, or the user settings, aren't easily handled now. The solution is to add a separate application-wide Controller, which handles application-wide commands and which stores the global UI state.

```
case EEikCmdHelpAbout:
    // Other common commands
```

```
iController->HandleCommandL(aCommand)
break;
```

All AppUi-specific Controllers pass unknown commands up to the application Controller, otherwise, `CAknViews` would need to know about both the application Controller and the view-specific Controller. Having a single Controller to cope with, the AppView is less tightly coupled to a specific application, increasing its chances for reuse across applications.

The controllers themselves will look identical to the ones in paper [2], so I will not repeat that code here. Check the accompanying code.

Creating the Controllers can be done when the AppUi is constructed, like:

```
void CPdfUIQAppUi::ConstructL()
{
    BaseConstructL();
    iController = CPdfController::NewL();

    iPageController = CPageController::NewL(iController);
    // Create the page view
    iPageController->SetPageView(iPageView);

    iFileController = CFileController::NewL(iController);
    // Create the file view
    iFileController->SetFileView(iFileView);

    // etc.
}
```

In this scheme, Controllers are not subclasses of each other and are not implementing the same interface. This isn't necessary, but it can be useful in some circumstances.

Consider two AppViews and two Controllers that handle the same commands. The object-oriented way of handling this case is to either subclass the two Controllers from a base class with possibly a common implementation, or to let the two Controllers implement a common interface. By keeping track of the current controller in an AppUI's member variable called `iCurrentController` with the base class type, setting this member when the current AppView changes, you can easily dispatch the command to the correct Controller.

Taking this idea further, there are further advantages in making all Controllers implement the same interface.

```
class MController
{
public:
    virtual void HandleCommandL(TInt aCommand) = 0;
};
```

A `CPdfController` implements the application-wide commands:

```

class CPdfController: public CBase
                    public MController
{
public:
    static CPdfController* NewL();
public:
    void HandleCommandL(TInt aCommand);
};

CPdfController::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
        case EEikCmdHelpAbout:
            AboutDialog::RunDlgLD();
            break;
        // Other application-wide commands
        default:
            // Ignore unknown commands
            break;
    }
}

```

And a CPageController handles the page-related commands:

```

class CPageController: public CBase
                    public MController
{
public:
    static CPageController* NewL(CPdfController* aAppWideController);

public:
    void HandleCommandL(TInt aCommand);

private:
    CPdfController* iAppWideController;
};

```

```

CPageController::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
    case EPdfCmdGoToPage:
        // Handle the Go To page command
        break;
        // Other page-related commands
    default:
        // Other commands are handled by the application-wide controller
        iAppWideController->HandleCommandL(aCommand);
        break;
    }
}

```

One advantage is that you can now code your AppUi's `HandleCommandL()` like this:

```

void CPdfUIQAppUi::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
        // Commands handled by the controller
    case EUIQSpecificCommand1:
        // Handle UIQ-specific command 1
        break;
    case EUIQSpecificCommand2:
        // Handle UIQ-specific command 2
        break;
        // etc.

    default:
        iCurrentController->HandleCommandL(aCommand);
        break;
    }
}

```

Reusing the file view?

At the beginning I mentioned that there should be reuse possibilities for the file view made possible by using a separate distinct Controller for this view. Unfortunately, there are more factors needed to enable reuse. As it happens, reusing the file controller wasn't as easy as I hoped it would be. Let's take a look why this is so.

In UIQ, file-based applications can use the `CQikMediaFile` class and friends for manipulating files. With the help of the UIQ Category mechanism, you get most of the tools needed for creating and managing a file view. The Media File classes do all kinds of convenient things for you, like showing progress dialogs and error messages.

The problem, however, should be quickly spotted from the names of the classes: this is strictly a UIQ-only solution.

In Avkon, you are on your own, at least, when you need to be compatible with Series 60 V0.9. Newer releases of the Series 60 Platform have introduced some classes helping you in this respect, but if your code must be compatible with the Nokia 7650, you have no other option than to write everything yourself.

In CKON/Series 90, there is the `CEikDirContentsListBox` class. Incompatible with both UIQ and Avkon UIs, this class has lots of functionality, mostly related to displaying and manipulating files in a folder.

The outcome for Pdf+ is that there is no reusable Controller class for files. It appears that hiding the differences between the three UIs is at least as much work as creating three different Controllers.

There are of course other options. One would be to use the Avkon implementation as a base for all three classes. However, that would mean that recreating the UIQ and CKON/Series 90 user experience might cost as much time, if not more, than creating the different Controllers.

In my opinion, the user experience must come first. A user must see identical behavior when doing the same thing in different applications. This is easiest to achieve by using UI-specific classes, when these exists.

In Pdf+, I chose to use three different Controllers. At least these Controllers and their associated file AppViews can be used as a base for creating other file based-apps, which is some comfort.

It would be interesting to see what interface these file views are going to present to the application and whether a single file view interface can be defined. As we have seen, the Controller interface is already known.

Using multiple controllers with a single AppView

If some function or functions cannot be made part of a Controller because these functions are not used in all distributions of the program, it is still possible to reuse the code in question. Instead of adding it to the Controller for the AppView, you can also create a distinct Controller for it, and reuse it in some AppUIs.

It's time for an example. Pdf+ has a search facility. In most UIs searching is managed entirely by your own code. In CKON/Series 80 however, there is a `CEikFindAndReplaceDialog` class which shows a find (and replace) dialog. In itself, such a dialog could have been hidden using the techniques described in paper [1]. Unfortunately this proved not to work. The dialog communicates its state using the well-known Observer pattern [3]. This means that the search Controller would have been the class implementing this interface, but then the search Controller could not have been used in the other UI implementations, because they don't have the interface. Again, I decided to use this built-in facility, to provide a consistent user experience. The final outcome was two classes, one generic search Controller that is in use in the Symbian OS v5, Avkon and UIQ distributions, and a separate one for the CKON UI.

Code using the separate controller will look like this:

```
void CPdfUIQAppUi::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
```

```

// Search
case EEikCmdSearchFind:
case EEikCmdSearchFindAgain:
    iSearchUi->HandleCommandL(aCommand);
    break;

case EUIQSpecificCommand1:
case EUIQSpecificCommand2:
    // Handle UIQ-specific command 1
    break;

// etc.
default:
    iCurrentController->HandleCommandL(aCommand);
    break;

```

Here, I called this class `CSearchUi`, to differentiate it from the larger page and file Controllers. The `CSearchUi` class has more in common with, for example, the built-in `CSendAppUi` and `CIrListenAppUi` classes.

You can create the `CSearchUI` class in the `AppUi`'s `ConstructL()`, or possibly when it is first called, like this:

```

case EEikCmdSearchFind:
case EEikCmdSearchFindAgain:
    if (!iSearchUi)
        iSearchUi = CSearchUi::NewL();
    iSearchUi->HandleCommandL(aCommand);
    break;

```

This trick of using multiple Controllers can be extended to the level of single commands. In such a case, you wouldn't think of using a Controller anymore, but you would be thinking in terms of implementing a command. Also, at this fine-grained level, you would not call a `HandleCommandL()` method, but name the class after the command. An example in Pdf+ is the command that exports the PDF file as an ASCII text file.

```

case EPdfExportAsText:
    ExportAsText::RunL();
    break;

```

The implementation would be similar to a method in a Controller, using interfaces that hide UI-dependent implementation details.

If you follow this idea to its logical conclusion, an `AppUi`'s `HandleCommandL()` would consist only of calls to one or two big Controllers, and a multitude of calls to small commands, some

commands being reusable and reused in multiple distributions, and the other commands being written for a single distribution.

Finally, note that this idea brings us very close to the Command pattern from Gamma et al. [3].

Conclusion

We have seen that it is possible to use a platform-independent controller class to implement a large part of the UI of a Symbian OS application. Using a single Controller however limited us to not reusing as much of the code as possible. By introducing more Controllers, more code could be reused.

There are two ways of adding Controllers to a program, firstly by giving each AppView its own Controller, and secondly by using multiple controllers for a given AppView. Both ways give opportunities for code reuse, both within multiple distributions of an application and between different applications for the same platform.

Background

Before becoming a Symbian OS C++ developer, Sander van der Wal worked in Pascal as a systems designer and project leader for a Dutch software company building, maintaining and selling a hospital information system. The programs were of the traditional, non-event driven kind. Some of the projects he worked on include a subset SQL compiler and an almost complete rewrite of the company's text editing system.

After establishing mBrain Software, Sander published four highly successful programs available to the general public: Pdf+ (an Adobe PDF file viewer), PdfPrinter (a PDF document creator), Fonts (a font file management tool) and FontMachine (an Open Font System font renderer). Sander has extensive experience with Symbian OS phones – Pdf+, for example, is available for all Symbian OS devices currently in the market.

References

[1] *Designing and building portable UIs for Symbian OS: How to design dialog interfaces*
http://www.symbian.com/developer/techlib/papers/cpp_programming_technique.html#portabledialogs

[2] *Designing and building portable UIs for Symbian OS: Using a controller as a portable UI component*
http://www.symbian.com/developer/techlib/papers/cpp_programming_technique.html#controller

[3] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns*. Addison-Wesley. ISBN: 0-201-63361-2.