

Designing and building portable UIs for Symbian OS: Using a controller as a portable UI component

Sander van der Wal, mBrain Software

Revision 1.0, February 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 isn't an obstacle anymore in the quest to writing a platform-independent UI layer.

This paper will build on the first one by describing how we can extend the actual implementation of a dialog to create an even larger platform-independent UI-layer. There is again lots of code in this paper. The accompanying ZIP file has this code, and an example of how to structure your source and build tree for an application designed using the methods presented in this paper.

First it's time to recap how a Symbian OS application is structured.

Application structure

Uikon provides a number of base classes from which you will (eventually) derive your own classes when writing an application:

- `CEikApplication` is the main application class
- `CEikDocument` is the document class
- `CEikAppUi` is the application UI class

In a CKON application, you will use these classes as the base class for your own application, document and AppUi classes. Furthermore, you will generally use a `CCoeControl` as the base class of an application view class.

Avkon introduced a number of new classes. Instead of deriving from the Uikon classes, in an Avkon application you should derive from `CAknApplication`, `CAknDocument` and `CAknAppUi` or `CAknViewAppUi`. An application view can be derived from a `CCoeControl` or additionally `CAknView`. It is also possible to use a dialog as an application view.

An application view class derived from `CAknView` is rather interesting. A `CAknView` gets the first shot on handling commands and it also has a menu. This means that such a class takes over some of the responsibilities of the AppUi class. As a result you'll see the `HandleCommandL()` and `DynInitMenuPaneL()` methods in a `CAknView`-derived class.

UIQ also has its own application document and AppUi classes too, in the form of `CQikApplication`, `CQikDocument` and `CQikAppUi`. UIQ doesn't introduce a separate application view class, so it is still common practice to derive such a class from `CCoeControl`.

Both Avkon and UIQ application views can implement the `MCoeView` interface. In Avkon, the `CAknView` class implements the interface directly, and it lets derived classes implement the interface through a set of its own virtual methods.

In UIQ, you must derive your application view from `MCoeView` yourself. Because most applications have two application views, a list view and a detail view, it is common practice to

base your own application views on a common base class, which is mainly used to share the implementation of the `MCoeView` interface, as far as possible.

Commonalities and differences

The problem here is this: much of the code of the main application classes will be identical between the different UIs. This is particularly so for the application, document and `AppUi` classes. If you have written or examined a number of Symbian OS applications you will have noticed that this is *especially* true for the application and document classes.

However, these two classes aren't the ones you will spend much time coding and debugging. They are so simple and straightforward that even a computer program can write them, as can be seen in the case of Avkon, using the Avkon application wizard.

The class where things are really happening is the `AppUi` class. For instance, all commands are routed through this class, using its `HandleCommandL()` method. Handling commands makes up the bulk of an application's code, so being able to reuse the `AppUi` class will result in major savings of both time and testing effort.

In Avkon you can also handle commands in an application view. It would be very nice if our method could also be used for `CAknViews`.

However, there are some real differences between UIs. Take for instance a file-based application on UIQ and CKON. CKON devices have a user-visible file system and users are expected to be able to navigate the file system. UIQ applications will present files using the category mechanism. The implementation is based on a file system, but the user is hardly ever expected to see it or use it.

Another difference between CKON and UIQ or Avkon is that CKON applications can print, whilst UIQ and Avkon applications are not expected to print at all.

This means that there are genuine differences between different distributions of your applications and that a method to reuse code must be able to cope with these differences.

Let's examine some ways of reusing code.

Copy the files and adapt them

Copying the files and adapting them will only work in the most straightforward of circumstances, for simple classes that are hardly ever or even never updated after they have been released.

This method is perfect for creating different implementations of the platform-specific application and document classes. Although the application class will be ready immediately, the document class might need some additional code.

Using macros

Another method to differentiate between different distributions is to use macros. An example is:

```
#ifdef UIQ_CODE
// Do something UIQ specific
#   define APPUI_PARENT_CLASS CQikAppUi
#elif CKON_CODE
// Do something CKON-specific
#   define APPUI_PARENT_CLASS CEikAppUi
#endif

class CMyAppUi: public APPUI_PARENT_CLASS
```

The advantage of this method is that it is quick and easy if there are just a few differences that need to be taken care of. However, the more complicated the code becomes, the harder it is to maintain such code, which is the method's main disadvantage.

I have used this method to create `CCoeControls` portable to both Symbian OS v5 and Symbian OS v6.0 and beyond. Symbian OS v5 has some differences in a number of `CCoeControl` methods. In particular some methods such as `SizeChanged()` were allowed to leave, and therefore had a trailing `L`. By making sure that the method doesn't leave, it is possible to use a common implementation like this

```
#ifndef _UNICODE // Signals Symbian OS v6.0 and later
void CMyControl::SizeChanged()
#else // Symbian OS v5 and earlier
void CMyControl::SizeChangedL()
#endif
{
    // Non-leaving implementation
}
```

Not the kind of code you'll want to see all over the place, I'd say, but for a few well-understood and isolated cases, it works fine.

Inheritance

A third method is to use inheritance to differentiate between UIs. At first sight, this appears to be the perfect solution. Object oriented programming was after all invented to enable people to 'program-by-difference'. You would create a base class that has all the common code, and you would then subclass the base class for each different specialization, in our case for each different UI. The base class can also store the UI state as member variables.

You can let either the document or the `AppUi` own the controller. Further, you can choose to let either the controller or the document own the model. In any case, you'll have a pointer to the document in the controller. I haven't shown this in the code, but it should not be hard to arrange this in your own programs.

An implementation, for instance for Pdf+, would look like:

```
// Base class with common commands
class CPdfAppUi: public CEikAppUi
{
public:
    void ConstructL();
    ~CPdfAppUi();
// --o CEikAppUi
protected:
    void HandleCommandL(TInt aCommand);
protected:
    // Common commands
    void CmdGoToPageL();
protected:
    // UI state variables
    TInt iPage; // Current page
    TInt iZoom; // Current zoom factor
    // Other common variables
```

```

};

void CPdfAppUi::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
        case EEikCmdHelpAbout:
            AboutDialog::RunDlgLD();
            break;
        case EPdfCmdGoToPage:
            CmdGoToPageL();
            break;
            // etc.
    }
}

void CPdfAppUi::CmdGoToPageL()
{
    TInt newPage = iPage;
    if (GoToPageDialog::RunDlgLD(newPage, iDocument->NumPages())
        {
            if (iPage != newPage)
                ChangePageL(newPage);
        }
}

```

Do notice that having a single interface for all our dialogs is vital here [1]. We could not have written this code otherwise.

The UIQ implementation can now be implemented like this:

```

// Base class with common commands
class CPdfUIQAppUi: public CPdfAppUi
{
public:
    void ConstructL();
    ~CPdfUIQAppUi();
// --o CPdfAppUi
public:
    void HandleCommandL(TInt aCommand);
// --o CPdfUIQAppUi
private:
    // UIQ-specific commands
    void CmdDeleteFileL();
    // etc.
};

```

```

void CPdfUIQAppUi::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
        // Deleting a file is a UIQ-specific command
    case EPdfCmdDeleteFile:
        CmdDeleteFileL();
        break;
        // etc.
        // The base class handles all common commands
    default:
        CPdfAppUi::HandleCommandL(aCommand)
        break;
    }
}

```

Similar code can be written for other device-specific commands. And, as has been said in the first paper of the series [1], you would still have to write CKON and UIQ-specific implementations of the dialogs. But writing the code for handling the common cases of an application would only have to be done once.

There are, however, some problems with this approach.

Firstly, in Avkon, you cannot have a CAknView class that is derived from CPdfAppUi. It must be derived from CAknView. This means that this solution cannot be used for Avkon applications with CAknView-derived classes.

Secondly, with inheritance, there must be a common base class. But the problem is that there is no such base class, at least, most of the time there isn't one. CKON AppUis are derived from CEikAppUi, but most UIQ AppUis will derive from CQikAppUi, and most Avkon AppUis will derive from either CAknAppUi or CAknViewAppUi. As a result, inheritance cannot be used, at least not easily.

There is a solution, and that is to use templates to make the base class a parameter. You'll write your AppUi like this:

```

template<class AppUi>
class CPdfAppUi: public AppUi
{
public:
    void ConstructL();
    ~CPdfAppUi();
    // --o CEikAppUi
public:
    void HandleCommandL(TInt aCommand);
private:
    // Common commands
    void CmdGoToPageL();
private:
    // UI state variables

```

```

TInt iPage; // Current page
TInt iZoom; // Current zoom factor
// Other state variables
};

```

with the implementation as above.

If you are into serious C++ trickery, this might appeal to you. Personally, I don't like this approach. You must know a lot about programming with templates, and learning this takes time, time that is better spent on improving the program.

There is another approach that is much simpler and doesn't have any of the two drawbacks mentioned here, or the problems related to templates.

Delegation

A better solution is to delegate the implementation of a class to a different class. Instead of implementing the handling of the common commands in the AppUi class, we will delegate this to another class. This new class is a kind of AppUi class, but only handles the common functionality of all AppUi classes. In addition, this class will also host the UI state in its member variables.

In the MVC pattern, the new class acts like the controller, so we'll call this class a CxxxController.

The AppUi class, and in Avkon the CAknView class, will handle the platform-specific commands, or commands that are common, but cannot be implemented in a general way.

Time for some code. The implementation of HandleCommandL() will look like:

```

void CPdfUIQAppUi::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
        // Commands handled by the controller
        case EEikCmdHelpAbout:
        case EPdfCmdGoToPage:
            // Other common commands
            iController->HandleCommandL(aCommand);
            break;
            // Deleting a file is a UIQ-specific command
        case EUIQPdfCmdDeleteFile:
            CmdDeleteFileL();
            break;
            // etc.
        default:
            break;
    }
}

```

The controller itself looks like:

```

class CPdfController: public CBase
{
public:

```

```

    CPdfController* NewL();
    ~CPdfController();
public:
    void HandleCommandL(TInt aCommand);
private:
    // Common commands
    void CmdGoToPageL();
private:
    // UI state variables
    TInt iPage; // Current page
    TInt iZoom; // Current zoom factor
    // Other state variables
};

void CPdfController::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
        case EEikCmdHelpAbout:
            AboutDialog::RunDlgLD();
            break;
        case EPdfCmdGoToPage:
            CmdGoToPageL();
            break;
        // etc.
        default:
            break;
    }
}

void CPdfController::CmdGoToPageL()
{
    TInt newPage = iPage;
    if (GoToPageDialog::RunDlgLD(newPage, iDocument->NumPages()))
    {
        if (iPage != newPage)
            ChangePageL(newPage);
    }
}

```

If you compare the inheritance and the controller solutions, you'll notice that there is little difference between the two. The main difference is that the controller is not derived from any of the main Eikon classes. This simple difference makes the class reusable in all UIs. It can even be used in classes that aren't AppUis at all. See the example code in the accompanying ZIP file.

Using a controller class is nothing new in Symbian OS programming. A well-known example is in the battleships example programs from both the *Professional Symbian Programming* [2] and *Symbian OS C++ for Mobile Phones* [3] books. I strongly suggest you study the controller in the latest book [3], because it also introduces the notion that a controller is the perfect place for storing and restoring the AppUi state.

Tying up some loose ends

There are still some problems that need to be solved. Firstly, what about the application view? The application view is another major component of a UI, and there are bound to be some issues that must be addressed.

Secondly, how to organize the resulting source files?

Let's take a look at the application view first.

The application view

In the MVC pattern, the controller's responsibility is to tell the model what changes must be made to its state, and to tell the view that it must redraw itself because the model has changed. In Pdf+ for instance, this happens when you go to a new page. The model must load the new page and the application view must then draw the new page. In code this could look like:

```
void CPdfController::ChangePageL(TInt aNewPage)
{
    iModel->SetPageL(aNewPage);
    iPage = aNewPage;
    iAppView->DrawNewPage();
}
```

But the application view is also dependent on the UI you are working with. In Avkon it can be a `CCoeControl`, a `CAknView` or even a dialog. In UIQ and CKON it's a `CCoeControl`. So how can you use a platform-dependent class in a platform-independent class?

If your application view is a `CCoeControl`, you can use a common base class like this:

```
CPdfAppView: public CCoeControl
{
public:
    // Constructing and destructing
public:
    // Methods called by the controller
    virtual void DrawNewPage() const = 0;
};
```

If you can have the same implementation for the different UIs, the methods can be non-virtual and non-abstract.

If you need your Avkon application view to derive from `CAknView`, you cannot use a common base class. My favorite solution is to use an interface class.

```
class MPdfAppViewControllerInterface
{
public:
    virtual void DrawNewPage() const = 0;
    // Other abstract methods
};
```

The controller talks to the application view through this interface. You need a simple method to set the page view:

```
class CPdfController: public CBase
{
    // Other methods
public:
    void SetPageView(MPdfAppViewControllerInterface* aAppView);
    // etc.
private:
    MPdfPageViewControllerInterface* iAppView;
    // etc.
};
```

The platform-specific application view must now implement the interface:

```
CPdfUIQAppView: public CCoeControl, public
MPdfAppViewControllerInterface
{
public:
    // Constructing and destructing
    // --o MPdfAppViewControllerInterface
public:
    void DrawNewPage() const;
    // etc.
};
```

In Avkon, it will look like this:

```
CPdfS60AppView: public CAknAppView, public
MPdfAppViewControllerInterface
{
public:
    // Constructing and destructing
    // --o MPdfAppViewControllerInterface
public:
    void DrawNewPage() const;
    // etc.
};
```

The best thing is that you can still use a common application view base class if you want to. Just derive the common base from `CCoeControl` *and* from `MPdfAppViewControllerInterface`.

Finally, you would also have this problem when deriving from a common `AppUi` base class. Fortunately this means that you can use the same solution.

Source file organization

With the application now consisting of a number of platform-dependent and platform-independent sources, how do we set up a source file structure that reflects the structure of an application?

The structure I am using calls for each application to have its own folder. In this folder are sub-folders for the platform-independent UI-level code, and a folder for each OS version the application will be running on. Each OS-version folder has then a standard structure, with folders for the aif components, for bmp files, for building the application, for building the release and for the platform-dependent source code.

This structure is flexible enough to accommodate most applications, and it can be easily extended, if the need arises. For instance, if you must create a Siemens SX1-specific variant of your application, you can add a SX1 folder with its subfolders in the /6.1/Series60/ folder.

The .dlls and .apps are built according to the device-specific .mmp files in the different group folders.

For example:

```
<Application folder>
  UI                : Platform-independent UI code and include files
  6.0                : Symbian OS v6.0
    9200             : For the Nokia 9200 Communicator series
      aif            : resource file and bitmap files for the .aif file
      data           : bitmaps
      group          : the .mmp and abld files
      release        : the .pkg file
      UI             : CKON-dependent UI sources
  6.1                : Symbian OS v6.1
    Series60         : For Series 60 v0.9 and v1.2 smartphones
      aif
      data
      group
      release
      UI
  7.0                : Symbian OS v7.0
    UIQ              : For UIQ v2.0 and v2.1 mediaphones
      aif
      data
      group
      release
      UI
```

A 9200-specific .mmp file would look like this:

```
// The Platform-independent UI-level code
SOURCEPATH ..\...\..\UI
SOURCE PdfController.cpp
// Other Platform-independent UI-level sources
// Platform-dependent code
SOURCEPATH ..\UI
SOURCE AboutDialog.cpp
SOURCE GoToPageDialog.cpp
```

```

SOURCE PdfCKONApplication.cpp
SOURCE PdfCKONAppUi.cpp
SOURCE PdfCKONDocument.cpp

USERINCLUDE <engine-include path>

// UI-level includes
USERINCLUDE ..\..\..\UI // Platform-independent code
USERINCLUDE ..\UI // Platform-dependent code

```

The accompanying ZIP file has an example of this file structure, populated with the files at the appropriate places. You are not meant to be able to compile this example, its purpose is simply to illustrate the required structure.

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. Because the controller class is independent of the UI, it is possible to use it in different AppUis and even in non-AppUi classes such as a `CAknView`. Because we were able to hide differences between UIs when using dialogs, these differences weren't important anymore, enabling us to create a unified controller.

What we have done is divide the UI part of an application into three layers. In the bottom layer are platform-dependent classes like the dialogs. By giving them a platform-independent interface, we were able to create a middle layer that is completely independent of the UI. This layer contains the controller, and an interface to the application view. Finally, the top layer consists of the platform-dependent main application classes, the application, document, AppUi and application view classes.

In the third paper of this series we'll look further into the use of controllers when using multiple application views in an application, and what to do when different distributions of your application do not need the complete set of functionality provided by a controller.

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] Martin Tasker et al. *Professional Symbian Programming*. Wrox Press Ltd. ISBN: 1-861003-03-X. <http://www.symbian.com/books/index.html>
- [3] Richard Harrison et al. *Symbian OS for Mobile Phones*. John Wiley & Sons Ltd. ISBN 0-470-85611-4. <http://www.symbian.com/books/index.html>