Symbian OS: Building Portable UI

Version 1.1
April 12, 2006

S Y M B I A N   O S

Legal Notice

Copyright © 2006 Nokia Corporation. All rights reserved.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1. Introduction 5
2. Building a Portable UI 6
   2.1 Macros 6
   2.2 Inheritance 7
   2.3 Delegation 9
3. Supporting Multiple Screen Sizes and Controls 11
   3.1 S60 Platform Scalable UI 11
   3.2 Screen Sizes and Resolutions 11
   3.3 Input Methods 12
   3.4 Softkeys 13
   3.5 Using Controls for UI Portability 13
4. Terms and Abbreviations 15
5. References 16
6. Evaluate This Resource 17

Change History

June 28, 2004	Version 1.0	Initial document release
April 12, 2006	Version 1.1	Terminology updates made and references to the scalable UI of the S60 platform added.

1. Introduction

This document presents the best practices for building a portable UI on Symbian OS.

Easy portability is important when introducing new UI implementations. This document presents practices such as macros, inheritance, and delegation as solutions for porting a UI. It also introduces scalable UI practices and discusses, for example, how to handle various device screen sizes, input methods such as key and pointer input, and the use of standard UI components for portability across various devices.

2. Building a Portable UI

Developing portable UIs is becoming an increasingly important issue as new UI implementations are introduced. It should be possible to reuse the code of the application in different platforms to the best extent feasible. The different practices for implementing a portable UI will be introduced later in this chapter.

The following practices will be discussed: macros, inheritance, and delegation, as described in *Designing and building portable UIs for Symbian OS: Using a controller as a portable UI component* [2] by Sander van der Wal.

Macros can be used to solve simple, small-scale portability issues in the application, but should not be used for handling complex and large porting issues because maintaining the macros becomes a time-consuming task.

Inheritance is used to build a common base class implementing the common functionality. UI-specific subclasses are inherited from the common base class and the UI-specific functionality is implemented into them.

Delegating the common functionality of the UIs to a separate class, a controller class, is quite an elegant practice of building a portable UI. The controller class will receive calls from the application UI class, which hosts the UI-specific functionality.

Generally it has been a good practice to separate the UI from the rest of the application functionality, that is, the engine. The UI part should only take care of the user interaction, whereas the engine should handle the core functionality of the application. Refer to *Symbian OS: Platform-Independent Engine Development* [3], Chapter 3, for more detailed information.

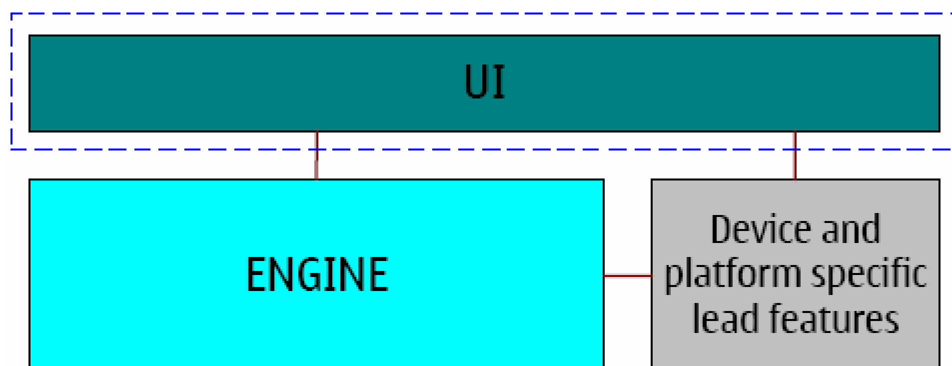

Figure 1: Separate UI and engine

2.1 Macros

Macros can be used in the code to manage the differences between UI implementations. Macros are useful as long as there are only a few differences to be handled. When the number of differences grows, more and more changes will be required to maintain the differences with macros, and the process will become complicated. Therefore, if you wish to use macros for portability, use them with moderation.

The following code example shows how to separate the `AppUi` class for UIQ [3] and AVKON [3] -specific implementations. The `AppUi` is inherited from different `AppUi` base classes, depending on the base class defined by the macro.

```
#ifdef AVKON_PORT
// definition for AVKON-specific AppUi class
#define APPUI_PARENT CAknAppUi
#elif UIQ_PORT
// definition for UIQ-specific AppUi class
#define APPUI_PARENT CQikAppUi
#endif

class CDemoAppUi : public APPUI_PARENT
    {
    // …
    }
```

2.2 Inheritance

In the inheritance method, the base class will contain all the common code. A subclass containing the UI-specific code is inherited. Individual subclasses will be created for each different UI.

The following code defines the class definition for the `CDemoInheritanceAppUi` class that will act as a common base class, containing the common functionality across all UIs. However, inheriting the subclasses from a common base class poses a problem because the `AppUi` base class is derived from different `AppUi` classes on different platforms: AVKON `AppUis` are derived from `CAknAppUi` and UIQ `AppUis` from `CQikAppUi`. Therefore, defining a common base class with a fixed parent AppUi class is not straightforward.

Macros are used to define unique `AppUi` parents for the specific platforms. The following macros define the `AppUi` parent class for AVKON and UIQ UIs.

```
#ifdef AVKON_PORT
// definition for AVKON-specific AppUi class
#define APPUI_PARENT CAknAppUi
#elif UIQ_PORT
// definition for UIQ-specific AppUi class
#define APPUI_PARENT CQikAppUi
#endif

// common base class
class CDemoInheritanceAppUi : public APPUI_PARENT
    {
    public:
        void ConstructL();
        ~CDemoInheritanceAppUi();
    protected:
        void HandleCommandL(TInt aCommand);
    private:
        void Common1();
        void Common2();
    }


void CDemoInheritanceAppUi::HandleCommandL(TInt aCommand)
    {
    switch (aCommand)
        {
        case ECmdCommon1:
            Common1();
            break;
        case ECmdCommon2:
            Common2();
            break;
        default:
            break;
```

```
        }
    }

void CDemoController::Common1()
    {
    // …
    }

void CDemoController::Common2()
    {
    // …
    }
```

Next, the UI-specific subclasses are defined, in this example for AVKON. These subclasses will inherit the common base class and implement the UI-specific functionality.

```
// AVKON-specific subclass
class CDemoInheritanceAKNAppUi : public CDemoInheritanceAppUi
    {
    public:
        void ConstructL();
        ~CDemoInheritanceAppUi();
    protected:
        void HandleCommandL(TInt aCommand);
    private:
        void AKNSpecific1();
        void AKNSpecific2();
    }

void CDemoInheritanceAKNAppUi::HandleCommandL(TInt aCommand)
    {
    switch (aCommand)
        {
        case ECmdAKNSpecific1:
            // AVKON-specific
            AKNSpecific1();
            break;
        case ECmdAKNSpecific2:
            // AVKON-specific
            AKNSpecific2();
            break;
        default:
            // pass the common functionality to the base class

            CDemoInheritanceAppUi::HandleCommandL(aCommand);
            break;
        }
    }

void CDemoInheritanceAKNAppUi::AKNSpecific1()
    {
    // …
    }

void CDemoInheritanceAKNAppUi::AKNSpecific2()
    {
    // …
    }
```

The subclasses for other UIs are defined in a similar way.

2.3 Delegation

Usually, command handling in Symbian applications is done in the `HandleCommandL()` method in the `AppUi` class. However, implementing all the command handling code in the `AppUi` class will most likely make the application specific to a certain UI.

Now, instead of implementing all the command handling in `AppUi`, the common, UI-independent part of the code is delegated to a separate class. The UI-specific commands will be handled by the `AppUi` class, which will call the delegate class to handle the common functionality. This is probably the most elegant solution for creating the implementations for different UIs.

The delegate class acts as a controller in the MVC design pattern. Therefore, the delegate class will be called a controller class, `CDemoController`.

The following code example shows the implementation of the `HandleCommand()` method of the `AppUi` class and the delegate class `CDemoController` hosting the functionality for handling the common commands.

```
void CDemoAKNAppUi::HandleCommandL(Tint aCommand)
    {
    switch (aCommand)
        {
        // AVKON-specific
        case ECmdAknSpecific1:
            AknSpecific1();
        // common functionality
        case ECmdCommon1:
            iController->HandleCommandL(aCommand);
            break;
        case ECmdCommon2:
            iController->HandleCommandL(aCommand);
            break;
        }
    }

class CDemoController : public CBase
    {
    public:
        CDemoController* NewL();
        ~CDemoController();
        HandleCommandL(Tint aCommand)
    private:
        Common1();
        Common2();
    };

void CDemoController::HandleCommandL(Tint aCommand)
    {
    switch (aCommand)
        {
        case ECmdCommon1:
            Common1();
            break;
        case ECmdCommon2:
            Common2();
            break;
        default:
            break;
        }
    }

void CDemoController::Common1()
    {
    // …
    }
```

```
void CDemoController::Common2()
    {
    // …
    }
```

3. Supporting Multiple Screen Sizes and Controls

When developing applications for a broad array of devices, issues such as various screen sizes and the differences among the input devices and methods must be kept in mind. The applications, especially graphical ones, should dynamically adapt to the various screen sizes. The different input methods, such as key and pointer input, should be taken into account.

Whereas the standard UIKON (and EIKON) components ensure UI portability, the platform-specific UI implementations, such as AVKON and CKON, are better used for UI scalability.

3.1 S60 Platform Scalable UI

In addition to different UI platforms, also the S60 platform (from 2nd Edition, Feature Pack 3 onwards) now supports multiple screen resolutions across different devices, including the portrait and landscape orientations in certain resolutions. Issues related to this concept called scalable UI, however, are out of scope of this document. For more information on the scalable UI, refer to the documents Introduction To The S60 Scalable UI [1] and S60 Platform: Scalable UI Support [5]. Useful code examples can be found in the document S60 Platform: Scalable Screen-Drawing How-To [4].

3.2 Screen Sizes and Resolutions

In order to achieve portability across different platforms, the application needs to adapt to different screen resolutions without the help of platform-specific features such as the S60 scalable UI. Instead, adaptation to different resolutions and screen sizes should be done using APIs and methods available in all target platforms, as presented in what follows.

The position or the size of the UI controls should never be hard coded in the view unless it is absolutely sure that the UI control will never change its size or position. The screen size can be requested with the `CEikAppUi::ApplicationRect()` method. This method will return a `TRect` structure that is the drawing area available for the application, including the space for toolbar, tool band, and title bar, if the application requires them. Note that this method returns the actual screen size, whereas the `CEikAppUi::ClientRect()` method returns the area that excludes the space for toolbar, tool band, and title bar.

The UI components are derived from the `CCoeControl` class, which acts as the base control class for other controls. The `CCoeControl` class will receive a `SizeChanged()` event when its size changes and the control should react by resizing its contents, that is, the other controls it contains. The size change that triggered the `SizeChanged()` event can be retrieved with the `Rect()` method call. A UI should therefore be built to react to `SizeChanged()` events and to resize the UI controls based on the size received from the `Rect()` method.

Different screen resolutions can be handled by overriding the `SizeChanged()` method in the view. The following code example shows the overridden `SizeChanged()` method that rearranges the image in the control. The view contents should be drawn according to the size of the inner rectangle, and therefore it is taken into account that the size of the image drawer can be bigger or smaller than the view drawing area. The image will be centered in the view and finally, the image drawer is notified of the size change.

```
void CDemoView::SizeChanged()
    {
    // get inner view rectangle
    TRect rect = iBorder.innerRect(Rect());
    // get the size of the drawing area, depending on the
    // available view size and image drawer size
    TSize size(Min(KImageDrawerWidth, rect.Width()),
               Min(KImageDrawerHeight, rect.Height()));
    // position in the center of the view, calculate the topleft
    // corner position for drawing area
    TPoint topleft(rect.iTl.iX+(rect.Width()-size.iWidth)/2,
                   rect.iTl.iY+(rect.Height()-size.iHeight)2);
    // notify the drawer of the size change
    iDrawer->SetRect(topleft, size);
    }
```

3.3 Input Methods

Devices have different input methods depending on the UI platform used. Building a portable application requires handling the key and pointer input devices. The pointer input device should be taken into account, even though all devices do not provide pointer input. Currently, S60 and Series 80 devices support key input, whereas the Nokia 7710 multimedia smartphone has a pointer input device. Note, however, that the situation may change in the future. The application should be able to adapt to the different input methods that the devices support.

The application should implement the `HandlePointerEventL()` method in order to respond to pointer events. This method is called whenever a pointer event occurs and will receive the type of the pointer event as well as the pointer coordinates. The pointer events are triggered for pointer down, up, and drag actions.

The pointer events are delivered to the application only if the device actually supports the pointer input device. However, the `HandlePointerEventL()` method may still be implemented in the application. This assures that the application will work also with the pointer input devices, providing portability across different input methods.

The following code example shows how to enable pointer events and handle the received pointer events. First, the application must enable pointer move and drag events in order to receive them. Also, pointer grab events should be claimed, because this ensures that all subsequent pointer events will be delivered to the control as well.

```
// enable pointer move and drag events
Window().PointerFilter(EPointerFilterMove|EPointerFilterDrag, 0);
// claim the pointer grab events
ClaimPointerGrab();

// pointer event handler
void CDemoAppView::HandlePointerEventL(const TPointerEvent&
aPointerEvent)
    {
    // get pointer coordinates
    TInt pointerX = aPointerEvent.iPosition.iX;
    TInt pointerY = aPointerEvent.iPosition.iY;
    switch (aPointerEvent.iType)
        {
        case EButton1Down:
            // handle pointer down event
            HandlePointerDown(pointerX, pointerY);
            break;
        case EButton1Up:
            // handle pointer up event
            HandlePointerUp(pointerX, pointerY);
            break;
        case EDrag:
```

```
            // handle pointer drag event
            HandlePointerDrag(pointerX, pointerY);
            break;
        }
    }

void CDemoAppView::HandlePointerDown(TInt pointerX, TInt pointerY)
    {
    // …
    }

void CDemoAppView::HandlePointerUp(TInt pointerX, TInt pointerY)
    {
    // …
    }

void CDemoAppView::HandlePointerDrag(TInt pointerX, TInt pointerY)
    {
    // …
    }
```

3.4 Softkeys

The various devices and UI implementations often contain a different number of softkeys. S60 devices contain two softkeys, Series 80 devices four softkeys, and Nokia 7710 three softkeys. Typically, S60 devices use predefined softkeys, such as R_AVKON_SOFTKEYS_OPTIONS_EXIT, which, when bound to a softkey, causes the application to exit when the softkey is pressed. The following describes a resource for an example softkey definition:

```
RESOURCE EIK_APP_INFO
    {
    cba = r_demo_cba;
    }

RESOURCE CBA r_demo_cba
    {
    buttons=
        {
        CBA_BUTTON
            {
            id=EEikCmdExit;
            txt="Exit";
            }
        };
    }
```

The different UI implementations use the same resource file format for defining the softkeys. Therefore, a common resource file can be used across UIs on the S60 and Series 80 platforms and Nokia 7710. However, the softkey actions will map into different key events on different UIs, and therefore a unique key event handler must be implemented for each UI.

3.5 Using Controls for UI Portability

The EIKON components provided the base UI layer for the application in Symbian OS v5. From Symbian OS v6.0 onwards EIKON has been replaced by the UIKON layer, providing essentially the same backward compatibility to EIKON. The naming of the classes in UIKON was not altered; they are still referred to with the `CEik` prefix as they were in EIKON. This section concentrates on the UIKON rather than EIKON components because they provide the current Symbian OS base UI layer.

The application UI portability can be well assured by building the UI with the standard UIKON components. UIKON components provide the basic building blocks for higher-level platforms, such as AVKON, CKON, and UIQ. Therefore, by relying on UIKON components when implementing the UI, the application UI should work on all Symbian devices that derive their UI platform from UIKON.

By using the UIKON components, the normal look-and-feel of the platform that would be created with the platform-specific UI components is not achieved. However, this ensures portability and a common UI look-and-feel across various devices.

Whereas the UIKON components provide better UI portability, the AVKON and CKON components improve UI scalability. AVKON and CKON components are optimized for the screen resolution supported by the device in the specific environment. The components will therefore scale automatically. Problems will arise when custom controls or bitmaps that do not guarantee automatic scaling are used. In addition to that, text input fields of different sizes across different UI implementations, such as the implementations on the Series 80 platform or Nokia 7710 vs. the S60 platform, may also pose a problem if not handled.

4. Terms and Abbreviations

Term or abbreviation	Meaning
AVKON	UI implementation on the S60 platform.
CKON	UI implementation on the Series 80 platform and Nokia 7710.
EIKON	Base Symbian UI component. EIKON consists of a framework with a set of controls and standard dialogs. EIKON has been replaced by UIKON in Symbian OS v6.0.
MVC	Model-View-Controller design pattern.
UIKON	Base Symbian UI layer providing a core for higher-level UI. S60 is built on top of this.
UIQ	Pointer (pen) based UI implementation. See http://www.uiq.com/.

5. References

[1] Introduction To The S60 Scalable UI, available at http://www.forum.nokia.com

[2] Sander van der Wal. *Designing and building portable UIs for Symbian OS: Using a controller as a portable UI component*. http://www.symbian.com/developer/techlib/papers/desbuildportui/desbuildportui.pdf

[3] Symbian OS: Platform-Independent Engine Development, available at http://www.forum.nokia.com

[4] S60 Platform: Scalable Screen-Drawing How-To, available at http://www.forum.nokia.com

[5] S60 Platform: Scalable UI Support, available at http://www.forum.nokia.com

6. Evaluate This Resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by rating this resource.