A Symbian OS C++ project template for multi-GUI applications

John Pagonis, Symbian Developer Network, June 2006

"Any problem in computer science can be solved with another layer of indirection. But that usually will create another problem."

- David John Wheeler

Maintaining the same code base for multiple GUIs unless done in a very disciplined way, is often expensive and frustrating. This article, demonstrates how one can pragmatically maintain almost the same application code base (having already separated the engine already) in a cost effective way, for three GUI variants on Symbian OS, namely for UIQ 2.x, Series 80 2.0 and S60 2.x.

There are other articles that demonstrate techniques of how to design components for multiple GUI variants [1, 2, 3]. This article deals with *how to layout your project in a manner that helps you maintain the application code between GUI variants*; so that most work is kept in sync between them, by means of sharing the same files.

Essentially, this article and associated code will give you the basis on which to start your multi-GUI app development from scratch!

Your project's design starts from the moment you think and lay out its directory structure.

It is expected that, should you chose to use the provided code, you will modify it and make it your own. By no means should the design of the provided project be a dogma or a constraint for your project. This code should be a very good start for your project and hopefully show you the path to follow. There will be features of GUI variants that you will discover later, which you may need to add to this design. In the mean time your app will be targeting multiple Symbian OS GUIs.

1 Your options

One way to do the maintenance between the GUI variants is to keep all code in the same project and retarget it by using pre-processor directives. Do Not do it!

This leads to unreadable code and bugs, since trying to build different variants will be frustrating and quite slow as pre-processor's constants will have to be changed. Usually in such cases, there is only one MMP file per project and to switch between different

variants is problematic as some features may be included or excluded due to forgotten `#ifdefs`.

You IDE will most likely hate it as well and be confused will parsing and colour coding your source.

You will also find that if you go down this dark alley, you may think that you are "saving" yourself from having to use an SCM to integrate between branches of your code, but what you will be actually doing is creating a lot of noise for your programming (since you will be faced with code that you don't about, at any given time). Moreover this will constrain you as to how you may want to vary the structure and even features of your application for different variants.

Another way is to keep your code base as three separate projects completely.

In this case, you need to use your favorite SCM system to maintain three forks of your project. Every time you add some new feature in the application UI and model(s) you will need to propagate and integrate those from whatever variant you chose to be working in.

Remember that if you don't always start the new features from the same variant, you will need to do reverse as well as forward integrations.

A third way is to not only separate your code between engine and UI, but also to introduce a GUI variant layer. Each variant layer should only include the parts of the source that are specific to that GUI layer irrespective of which logic unit they belong to.

Since you will want to offer (almost) the same features in all three GUIs of your application, your `AppUi` and engine code should be common among the different variants anyway. Having said that, we know that, the AppUI code will need to inherit from `Ckon`, `Avkon` and `Qikon`, for the Series 80, S60 and UIQ variants respectively. So how can we achieve this?

2 Variant layer

Well, first we have to separate the interface from the implementation and then discover commonality and variability which we need to isolate in our implementation within the variant layer.

A directory structure that allows us to do so, looks like:

```
project/inc
      /inc/UIQ
      /inc/S60
      /inc/S80
      /src
      /src/UIQ
      /src/S60
```

```
      /src/S80
      /group
      /sis
```

Then we need to make a decision on how we will keep the AppUi header variants. There are two options

a) Keep them in one header in the common area of /inc and make it inherit from a mock/indirection class that will respectively need to keep a header in each variant layer which will need to inherit from Ckon, Avkon or Qikon [4, 5, 2, 3].

b) Keep three copies (yes 3 copies) of the variant AppUis in the three variant layers. Although this may sound quite amateur, do not dismiss it quickly. This is because it allows you to speciate your AppUi as much as you like in terms of the private members that you may want to have, as well as give you the freedom to add functionality that may not be found in other variants. The cost of doing so is the cost of maintaining the header between your AppUis manually (which will be forced on you by the common code anyway)

In both of the above cases you will need to have one MMP file per variant, which is a very good choice (as opposed to the single MMP with #ifdefs) because it allows you to quickly and iteratively build your projects as you go along - even doing so simultaneously, without touching any 'configuration definitions'.

Following this, your /project/group directory should look like:

```
/project/group/appUIQ.mmp
        /group/appS60.mmp
        /group/appS80.mmp
```

Each MMP file should then include the correct libraries and locations of your variant and common files.

For example, all three (pre v9.x) MMPs will have the same common sections such as:

```
TARGET project.app
TARGETTYPE            app
UID                   0xYYYYYYY 0xZZZZZZZZ
TARGETPATH            \system\apps\project
```

```
USERINCLUDE           ..\inc
SYSTEMINCLUDE         \EPOC32\INCLUDE
SYSTEMINCLUDE         \project\inc
SYSTEMINCLUDE         \someEngine\user
```

and variant sections in each MMP, such as

```
USERINCLUDE           ..\inc\UIQ
SOURCEPATH            ..\src\UIQ

...

USERINCLUDE           ..\inc\S80
SOURCEPATH            ..\src\S80

...

USERINCLUDE           ..\inc\S60
SOURCEPATH            ..\src\S60
```

Having done this, you need to decide which source files will go in the variant layer.
In the example project the layout of files looks like:

```
Common                <src>                   Variant
-------------------------------------------------
projectAppUi.cpp                  variantAppUi.cpp
projectBaseView.cpp
projectBaseControl.cpp
projectDocument.cpp
projectModel.cpp
projectEngine.cpp
projectApplication.cpp
                                  projectView.cpp
                                  project.rss
```

```
Common                <inc>                    Variant
-----------------------------------------------------
projectApp.hrh
projectBaseControl.h
projectBaseView.h
projectModel.h
projectEngine.h
projectView.h
project.hrh
                                    projectApplication.h
                                    projectAppUi.h
                                    projectDocument.h
```

What the above layout tries to do is to:

a) keep the interfaces common as much as possible

b) speciate the interfaces for `AppUi`, application and document because they need to inherit from a different GUI variant

c) speciate the parts (or whole) of views that are inherently GUI variant specific

d) separate the variant specific parts of the main `AppUI` away from the common ones and keep the common engine and application code.

3 AppUi

As illustrated in the example project, your `AppUi` implementation will need to be separated between the common and variant layers.

Why?

Well, because you will need to do

a) variant specific initialisations

b) variant specific command handling (perhaps)

c) variant specific pointer handling

d) categories in UIQ (maybe)

e) variant specific control setup and handling

f) variant specific dynamic menu initialisations

Nevertheless, following this separation you'll have almost all of the `AppUi` code common between all three GUIs! No `#ifdefs` and no *diffing* will be necessary.

Therefore your MMP, for Series 80 for example, will look like:

```
SOURCEPATH          ..\src
SOURCE              projectAppUi.cpp
...
SOURCEPATH          ..\src\S80
SOURCE              s80AppUi.cpp
```

Where the implementation of the `CEikAppUi` derived `AppUi` of your project is defined between both the `projectAppUi.cpp` and the `S80appui.cpp`. In the latter you should keep all methods which are specific to the S80 variant.

In the case of the example project these are:

```
void ConstructUiVariantL()
void VariantHandleCommandL(TInt aCommand)
void VariantDynInitMenuPaneL(Tint aMenuId, CEikMenuPane*
      aMenuPane)
void VariantDimButtons()
```

as well as the app document constructor

In fact, you may chose to add more or use different naming. Realistically you will need a variant specific method for handling some commands, or for dynamic menu pane initialisation and definitely for the construction of the `AppUi`. You may also need to add variant methods for your document storage and retrieval.

4 Resources

As shown in the project layout above, resource files have to be in the variant layer since they define GUI variant features that may not exist between GUIs. Subsequent resource files that link from the main one may be put in the common layer as long as the components that make use of them have been architected that way.

5 A View to a switch

5.1 AppUi and Views

For each GUI variant of your application you will need to derive your `AppUi` from a different application UI framework. This may seem obvious and trivial and has been documented before [6,7]. Apart from a caveat or two!

On S60 the choice of `AppUi` derivation influences the view architecture employed. In turn this influences the class design of your application's views, view switching and to a certain extend that of the main `AppUi` component.

In S60 you have two choices:

a) Implement your `AppUi` by deriving from `CAknAppUi`

b) or implement your `AppUi` by deriving from `CAknViewAppUi`

Doing the latter, forces you to implement your views not in terms of dynamic binding on the `MCoeView` interface and derivation from `CCoeControl`, but instead, in terms of derivation from the concrete `CAknView;` which indeed inherits the protocol declared by `MVCoeView`. If you do so, then you will need to use aggregation in order to contain controls (deriving from `CCoeControl`) in your `CAknView` derived view

Your design decision will come down to the trade-off between gaining some `AppUi`-like functionality in your views (with fewer methods to call) and that of simply maintaining cross-variant compatibility.

In the example project the author chose the merits of simple compatibility.

On the Series 80 platform you will have to derive your `AppUi` from `CEikAppUi` (your application from `CEikApplication` and your document from `CEikDocument`), but you don't have to employ view switching (strictly speaking you don't have to do so on S60 and UIQ either).

In fact in most examples that you will come across, you will find that the paradigm used is that of calling views, which are in fact just composite `CONE` controls. This confuses many new developers. Examples and applications for Series 80 2.0 that you may have come across don't usually employ view switching due to legacy reasons since the Nokia 9210 didn't make use of views and neither did any 3rd party applications that were developed for it.

These days, we can use view switching on the series 80 2.0 platform, like the example project illustrates. Doing so makes porting from and to UIQ especially easy. Porting from/to S60 is a bit more difficult due to the limited screen real estate and use of soft keys as opposed to menu bars [8].

5.2 Views and variants

You will see that, in the example project, views have been completely isolated in their respective variant layer. This is not necessary but is a good start. If you feel that in your case you can sensibly reuse code between such views, then refactor it and move it into the common layer. In doing so though, you will still need to deal with view presentation, which is GUI specific.

If you find yourself discovering a lot of code, which happens to be common, between the same (logical) view across different GUI variants, you should ask if such code belongs logically at all in the view! Can it be that such code belongs to some other aspect of the MVC (or MVP) pattern? [9, 10, 11]. If so then refactor it and again keep your views thin and presentation specific. In the case where your design follows the MVP [12] paradigms, it is then when you can most likely separate part of your view(s) between variant and common layers.

Having said that, in Symbian OS C++, classes that implement the `MCoeView` view switching protocol act as containers, as well as context for controls; that in fact implement numerous MVC based components. To keep things straightforward the example project, goes nowhere near such subjects.

6 Other tricks

6.1 Common control base

As demonstrated in the example project code, you may also want to move some commonality in a base class for all the controls in your project. Although this idiom is not directly related to multiple GUI projects per se, doing so will most likely make reuse and handling of controls simpler

Your views will also need to inherit such common functionality from your control base class as well, employing more re-use.

6.2 Common view base

Another technique which some industrious developers may chose to employ is that of separating the views in common and variant layers by another level of indirection, thus sharing the same code between different GUI variants.

Doing so is not easy, since you will find yourself battling with little differences between GUIs and screen real estate management, which in most cases complicates the original design. Having said that, it is something that may prove to be useful for maintenance, after few revisions of the application, when the code base has been stabilised and the commonalities may be refactored.

7 To conclude

Use the Multi-GUI application template project provided and start building your application from there while refactoring whatever you feel necessary. It has been tested on all three variants and its layout will make you keep your main application logic in one place, which will speed up your development and make maintenance much simpler.

8 References

[1] Designing and building portable UIs for Symbian OS: How to design dialog interfaces, Sander Van der Wal

[2] Designing and building portable UIs for Symbian OS: Using a controller as a portable UI component, Sander Van der Wal

[3] Using an indirection DLL for multi-UI platform support, Andy Weinsein

[4] Advanced C++ Programming Styles and Idioms, Coplien, J. O.

[5] Design Patterns: Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson and J. Vlissides.

[6] Symbian OS C++ for Mobile Phones vol2, Richard Harrison

[7] Series 60 Developer Platform: Porting from UIQ

[8] Series 80 Developer Platform: Porting From Series 60

[9] "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk -80", Glenn Krasner and Stephen Pope

[10] Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC), Steve Burbeck

[11] Object Oriented Programming - An Evolutionary Approach, by Brad Cox

[12] MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java, Mike Potel