

Getting Started with Nokia's Carbide.vs 2.0 Development Tools

a tutorial by Andreas Jakl / Mopius

1. Introduction

The decision which IDE to use for Symbian OS / C++ development hasn't been easy in the last few years. Microsoft Visual Studio 6 lacks comfort, VS.net was not properly supported, Borland C++ BuilderX Mobile Edition and Metrowerks CodeWarrior were not to everyone's liking. Fortunately, this situation has changed recently with the introduction of Nokia's Developer Suite for Visual Studio .net.

A short time ago, the successor has been released, already bearing the name of the next generation of Eclipse-based Symbian OS tools that are to be released in 2006: *Carbide*.

While Carbide.vs doesn't provide tools like a visual UI designer, it does hide most of the complex handling of Symbian OS SDKs and their command line tools. With additional and very helpful pre-written code segments, it integrates Symbian OS development nicely into the IDE.

This tutorial is aimed at people who want to start with mobile development for Symbian OS or have been using other tools before and would like to get an overview of the new possibilities of Carbide.vs. It's a good time to try it out, as Symbian is now slowly opening its platform to the mid-tier market. With the limitation of servicing only the highest-end smartphones gone, cheaper devices will appear, which appeal to even more people.



Figure 1: We're going to use the helpful features of Carbide.vs to modify a HelloWorld application with just a few clicks and even less lines of code

2. Preparations

To work through this tutorial, you need the following software – make sure that you install it in the specified order to prevent possible problems:

- **Microsoft Visual Studio .net 2003:** During installation, it will ask you about setting system path variables. It's recommended that you allow this.
- **Java JRE:** Always choose the latest version, which can be downloaded from:
<http://java.sun.com/>
- **ActivePerl:** As before, make sure that you allow the installation program to set the path variable. Download it from: <http://www.activestate.com/Products/ActivePerl/>
- **Symbian OS Series 60 SDK:** Which one you need depends on the device that you would like to target. Check your requirements at <http://www.forum.nokia.com/devices> and download the appropriate SDK from the same site. This tutorial is aimed at one of the Series 60 2nd Edition SDKs.
- **Carbide.vs:** Download it from the same site as the SDK (Forum Nokia). It's helpful if you download the help files as well, so that they are directly integrated in Visual Studio .net.

One of the more common error messages that you might encounter when compiling your projects is “Error Spawning CL.exe”. To prevent this from happening, add the following paths to the VC++ directories of Visual Studio (*Tools->Options->Projects->VC++ Directories->Executable Files*):

C:\Symbian\7.0s\Series60_v21\Epoc32\tools, C:\Program Files\Microsoft Visual Studio .NET 2003\Vc\bin, C:\Program Files\Microsoft Visual Studio .NET 2003\Common\IDE (If necessary, adapt the paths to your corresponding installation directories).

3. Get Started

We'll dive right into creating your first project with Carbide.vs. In Visual Studio, start a new project of the type *New Symbian OS Project* and give it the name "HelloWorld" (see Figure 2). It's recommended that you choose the directory *C:\Symbian\dev* to place your projects, so that they are close to the SDKs. Make sure that you don't have any spaces in your project directory names, as those cause problems for the command line-based SDK tools. Don't worry, normally you do not have to work with them yourself, but they are still executed automatically by Carbide.vs.

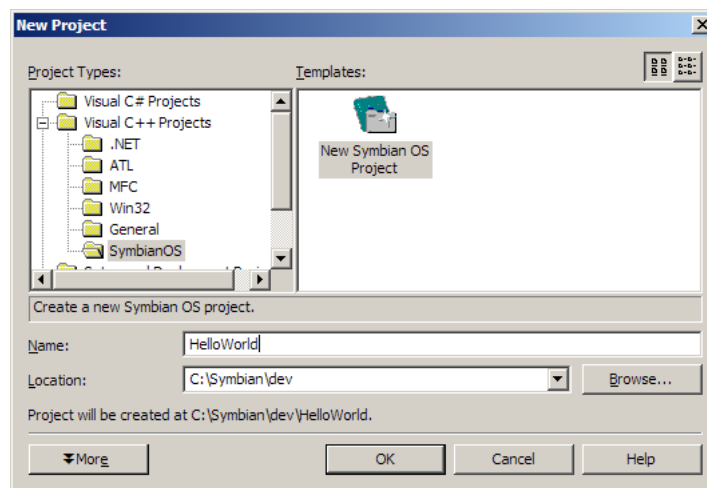


Figure 2: Starting the Carbide.vs wizard for creating a new Symbian OS project

Next, you will get to the settings page of your new Series 60 project (see Figure 3). Choose the type *Classic* - we're creating a Symbian OS 7 or 8 application, so that we can actually try it out on existing devices. Symbian OS 9 will introduce some fundamental changes and is not binary compatible with previous versions of the operating system. This is necessary to make room for significant improvements and has happened for the first time since Symbian OS mobile phones were available. If standard functions were used, applications had been working on all phones without any changes up to now. For example, the Symbian OS game "The Journey" is working perfectly fine on the old Nokia 7650, the Siemens SX1 as well as the brand new Nokia N90 – without any device-specific modifications. Usually, J2ME developers are rather jealous when they hear about this...

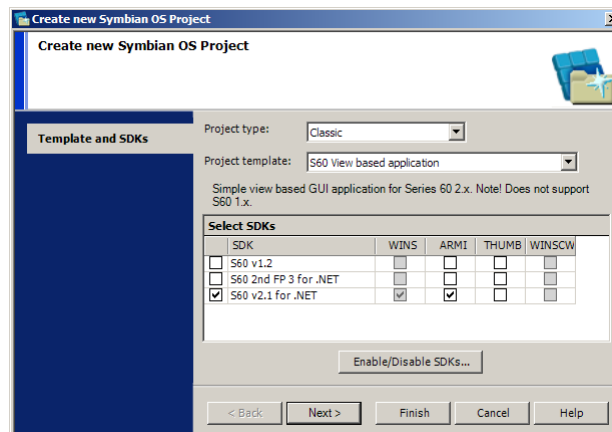


Figure 3: Choose the SDK you're interested in and create a view-based application

For the rest of the project settings, you only have to change two things. First, activate ARMI for the SDK you want to work with. WINS is needed when you build your application for the Windows emulator. This is what you'll do most of the times, as it features full debugging support. ARMI refers to the ARM-chip used in Symbian OS phones. Second, choose a View-based application. We could create a simple *Hello World* example, but the View-based project template comes with a more flexible layout and class structure.

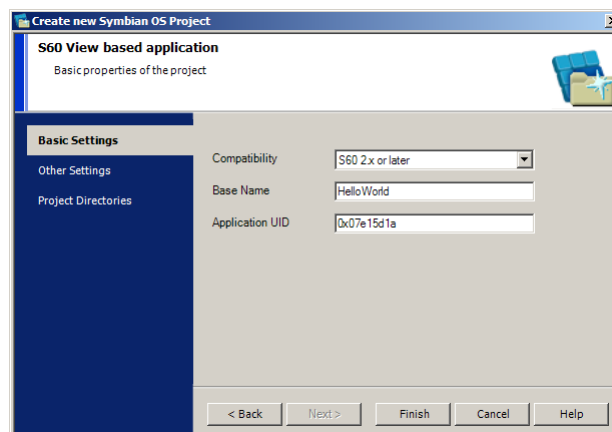


Figure 4: The application UID is important when you plan to release your program

On the following page, the wizard will ask for a program UID (see Figure 4). This is a unique identification number, each application on the phone has to have a different number. You can simply leave the default value in there, as the application was automatically assigned with a random

development ID. If you want to release your application, you have to apply for a real UID at <http://www.symbiansigned.com>.

To test if everything worked well, make sure that the *Solution Configuration* (next to the play button) is set to a debug build for the emulator (WINS). Simply press [F5] (don't use [Ctrl+F5]) and the emulator will launch. It behaves like a normal phone, so move down in the menu until you find your application. Use the buttons instead of clicking on the phone screen – it does partly work, but Series 60 screens do not have a touch screen. You'll notice that the wizard has already created a fully working graphical application with a menu and two views for you. This explains why there need to be so many source files. See Figure 5 for a screen shot.



Figure 5: This is what our HelloWorld-application looks like before we modify parts of it

If you want to try the application on your phone, change the build target to the release build for ARMI (for example *Rel_S60_21_ARMI*) and start the build process. A *HelloWorld.sis*-file will appear in the *sis*-directory of your project, which you can simply send to your phone using Bluetooth or your PC Suite.

4. Graphics

Next, we'll display a simple bitmap on the screen. First, draw a small picture and save it to the *data*-directory of your project (*C:\Symbian\dev\HelloWorld\data*) using the name *Picture.bmp*. You can also do this directly through Visual Studio. Right-click on the *data*-directory in the solution explorer and choose *Add->Add New Item...->Bitmap File (.bmp)*

Once you are tired of painting the most beautiful picture the world has ever seen, we'll tell Symbian OS to put it into an *.mbm*-file. This is a bitmap collection, which can handle multiple bitmaps and makes it easy for you to access them from within your program. Even better: Carbide.vs helps with this task.

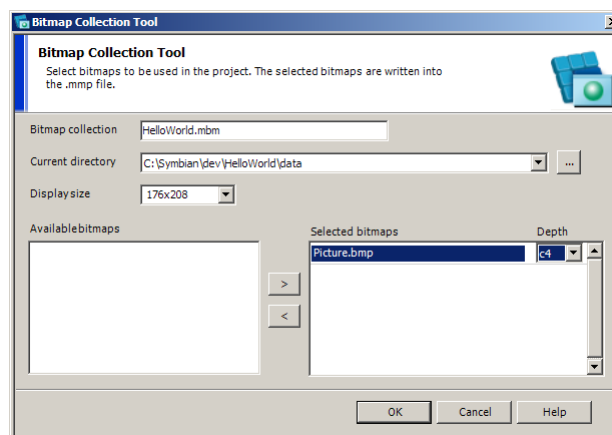


Figure 6: The Bitmap Collection Tool helps you with combining multiple pictures into one file

Simply right-click the *HelloWorld.mmp*-file in the Solution Explorer and start the *Bitmap Collection Tool* (Figure 6). Set the display size to 176x208 (which is the default for most of the current Series 60 devices), go to the *data*-directory and add your bitmap to the right side, setting its colour depth to *c8* to have an 8-bit colour image. Now, compile the project for the mobile phone target again, so that the build tool chain creates the required files.

4.1. Finding the Picture

Of course you won't see the picture in your application yet, we'll have to write some code to load and display it. How to find and reference the picture? Search for *HelloWorld.mbg* in the directory of your Symbian SDK. By opening it with a text editor, you will find the following text:

```
enum TMbmHelloworld
{
    EMbmHelloworldPicture
};
```

We have added one file called *Picture.bmp* to the bitmap collection, therefore the enum only has one ID, which includes the name of your drawing. Now that we know the ID of the picture, the only thing left to do is to display it! That's not difficult, but we'll do it the trial-and-error way so that you can see how errors are reported and how to find solutions.

As mentioned before, your bitmap collection is called *HelloWorld.mbm* – and that's exactly the file we want to load. Go to *HelloWorldContainer.h* and add the following line above the class definition:

```
#include "HelloWorld.mbg"
_LIT(KMbmFileName, "HelloWorld.mbm");
```

The command `_LIT` is actually a predefined macro that creates some kind of constant string with the name `KMbmFileName`. To talk using the language of Symbian OS, it's called a descriptor, not a string. Symbian OS phones have a normal file system like Personal Computers, so the file name won't be enough to find it. We'll have to add the path where the application is installed to the file name. To do that, go to `ConstructL()` in *HelloWorldContainer.cpp*. This is where the following code belongs:

```
TFileName fullName(KMbmFileName);
CompleteWithAppPath(fullName);
```

This creates a standard modifiable descriptor object called `fullName`, containing our file name. The next line adds the application path to the text, as you might have guessed by looking at the name of the function. Sounds good, so let's try if it works so far. Make sure the WINS Debug-variant is set as the build target and compile the project.

Unfortunately, the result is our first error - `CompleteWithAppPath` wasn't found. If you have installed the SDK-help for VS.net, go to *Help->Search...* and let your computer search for the missing

function. Make sure that the filter is set to “(no filter)” instead of the MSDN libraries. The results will contain a lot of references to a file called `aknutils.h` – this is the one we have to include in the `HelloWorldContainer.h` file (`#include "aknutils.h"`). This solves our first problem.

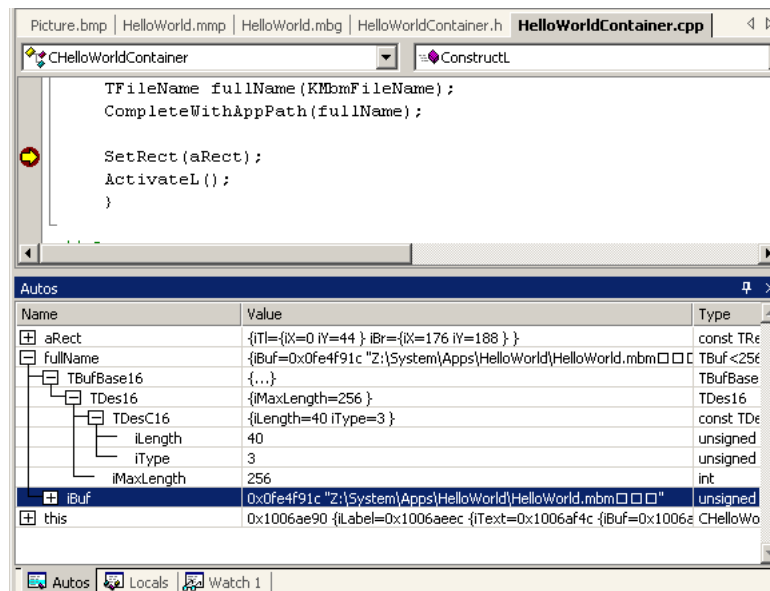


Figure 7: Debugging reveals the full path of the .mbm-file

If you'd like to know what's added to the file name, simply place a breakpoint in the corresponding line of your code and start your application in the emulator. By expanding the variable name `fullName`, you'll discover the whole path in the `iBuf`-subcomponent (Figure 7). If you don't see this anywhere on your screen, activate the window through `Debug->Windows->Autos`. In the emulator, the drive is reported as `Z`. That's actually the ROM-drive of the mobile phone. If you install your application on your real device, it will usually be on `C` (internal memory) or `E` (optional memory card).

4.2. Now let's display it!

All you need to do is adding four lines to your code. First, define a new private instance variable that stores a reference to the image object in `HelloWorldContainer.h`:

```
CFbsBitmap* iImage;
```

The class `CFbsBitmap` is provided by the SDK and contains a bunch of useful routines for manipulating images. Next, go back to the `ConstructL()` function of the corresponding `.cpp`-file and add the following below the previously added text:

```
iImage = new (ELeave) CFbsBitmap();
User::LeaveIfError(iImage->Load(fullName, EMbmHelloWorldPicture));
```

Those lines do all the work for us. First, an instance of the object is created, which then loads the image from our *.mbm*-file with one simple call. Ignore the “leave”-stuff for now – it's the Symbian OS-way of handling errors.

You might have guessed where to draw the image if you took a look at the source code of the Container class – this piece of code has to be placed at the end of the `Draw()` function:

```
gc.BitBlt( TPoint(0,0), iImage );
```

With this call, your image is copied to the graphics context (`gc`) of the application screen at the upper left corner. Unfortunately, the following error will come up when you compile the project: “Unresolved external symbol: [...]CFbsBitmap::[...]”. By now you should already know that you can find the header file and (this time also) the required library by searching the SDK documentation. The include you need is `#include <fbs.h>`. Libraries that your project depends on have to be defined in the project definition file (*.mmp*). With Carbide.vs, you no longer have to edit this file manually – Nokia integrated everything nicely into VS.net. Make sure that the WINS-debug build configuration is active and go to *Project Properties->Linker->Input->Additional Dependencies*. Once there, add: *fbscli.lib* at the end of the list. That's it.

4.3.Memory Leaks

The good news is that the image is now showing up as it should! The bad news isn't directly visible and you will only notice it when you close your application in the emulator. The message “Program Closed: HelloWorld” isn't there to inform you that you closed your program, it's a serious error message that notifies you about a problem. However, in this form it doesn't tell you a lot about the cause of the error. A simple trick reveals what's really happening: Create an empty file called *ErrRd* in the folder `\Epoc32\wins\c\system\bootdata` of your SDK. When you restart the emulator and close your program, an additional text is added to the message: “ALLOC: [...]” (see Figure 8).



Figure 8: One of the error messages that will leave you wondering – if you don't know where to get further information

Finding the meaning of this error is easy if you know where to search. Go to *Help->Search...* once again. Look for “system panic reference” and activate the option “Search in titles only”. You will find a page that contains links to all kinds of error categories, including our “ALLOC:” error. Another thing you might want to take a look at is the “KERN-EXEC 3” panic, which is maybe the most frequently experienced problem. In nearly all cases, this is a NULL-pointer exception.

Back to our current problem. As every C++ developer will know, we have to delete the image object again in the destructor of our class. Do that as usual by:

```
delete iImage;
```

Now your program is working perfectly fine and you've learned quite a bit about finding information about errors using the integrated documentation.

5. User Interface

After dealing with bitmap graphics, we are going to dive into the powerful API's for creating user interfaces. Carbide.vs provides support for several input- and information dialogues. That's rather helpful as you have to manually add code to resource files. To demonstrate a simple use case, we will add a login-screen to our HelloWorld-application.

5.1. Extending the menu

The user should be able to select the Login-command in the Options-menu of the left softkey. Start with defining the text of the menu item in *HelloWorld.loc*:

```
#define qtn_view1_login_item "Login"
```

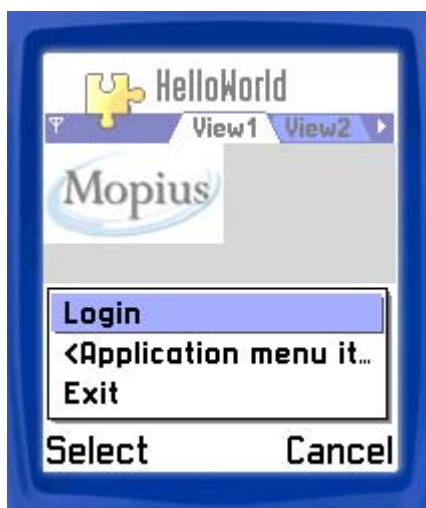


Figure 9: This is what our new menu item looks like

Next, go to *HelloWorld.hrb* and add a new command to the enumeration. After you're finished, the corresponding part of the file should look like this:

```
enum THelloWorldCommandIds
{
    EHelloWorldCmdAppTest = 1,
    EHelloWorldCmdLogin
};
```

The popup-menu itself is defined in *HelloWorld.rss*. We'll add the new item to the View1-specific menu, so that the new login command is only visible when the user is in View1. The according resource is called `r_helloworld_view1_menu`. We don't need the test-command the wizard created anymore, so replace `EHelloWorldCmdAppTest` with our new command `EHelloWorldCmdLogin`. Instead of the old text `qtn_view1_option_item`, use the name of our new text: `qtn_view1_login_item`. After this step, the menu item is visible (Figure 9). However, no action has been associated with it yet. We'll do that soon.

5.2. Input Dialog

As you can see in the resource file, the elements of user interface definitions can be difficult to learn by heart. Luckily, we have got Carbide.vs, which can write the code for us. At the end of the resource file, choose *Add Fragment...* from the right-click menu. The *S60 Multiline query dialog* looks just like it was made for this tutorial. On the next page, change the general name to something more meaningful, in our case `r_login_query` would be a good idea. A moment later, the complete resource definition appears in your file. Above, you'll even see instructions on how to include this in your application!

The two labels of the dialogue ("Username:" and "Password:") are specified directly in the definition. For a perfectly structured application like our HelloWorld program, this isn't what we want. Like the rest of the text definitions, we will move those two strings to the *.loc*-file. This has the advantage that the text is not spread through your whole application. It's also necessary for localization.

Add the following two lines to the *HelloWorld.loc*-file:

```
#define qtn_username "Username:"  
#define qtn_password "Password:"
```

Then replace the text (including the quotes) in the dialogue definition in *HelloWorld.rss* with the new names that we just assigned (`qtn_username` and `qtn_password`).

For adding the box to the program, we can simply follow the instructions from Carbide.vs. Start with adding a new function to *HelloWorldView.h* in the private function declaration block:

```
void HandleLoginL();
```

To call this function when the user selects the new menu item, add this code to the switch in `CHelloWorldView::HandleCommandL()`:

```
case EHelloWorldCmdLogin:
{
    HandleLoginL();
    break;
}
```

Right now, our view neither knows the name of our new menu command, nor has it heard about how to use a query dialogue. To change that, add the following includes to *HelloWorldView.h*:

```
#include <AknQueryDialog.h>
#include "HelloWorld.hrh"
```

As the instructions suggest, add the library *avkon.lib* to the project properties. To finally finish this work package, copy the code for displaying the dialogue box to a new `HandleLoginL()`-function. See the final code in Listing 1 for a first reference. If you didn't forget one of the steps, a full-blown dialogue box with complete autonomous text input handling should appear when you select the new Login menu item (Figure 10)!



Figure 10: A complete login box can be displayed with just a few lines of code

5.3. Checking the login

The code prepared by Carbide.vs already differentiates between the user cancelling the login attempt and closing it the way it was meant to be. As we don't want everyone to access our top secret data just because he doesn't cancel our great login dialogue in panic, we have to check and verify the user input somehow.

To let the application know what's correct, add the following definition at the top of

```
HandleLoginL():
```

```
    _LIT(KCorrectLogin, "mopius");  
    _LIT(KCorrectPwd, "asdf");
```

Of course you can choose your own login user name and password. Please note that you shouldn't write highly confidential information into your application this way. The text will be stored in the compiled application file without compression of any kind, making it possible to read it without too much hacking knowledge.

In the code for displaying the login box, you should rename the variable `value1` to `username` and `value2` to `password`. They are of the type `TBuf` with the maximum length set to 8. A `TBuf` is a modifiable descriptor, you might remember that this is the Symbian OS way of dealing with strings. The class provides a useful set of routines for working with text content. In our case we need the `Compare()`-function. But before including this, we should take a look at how to display a beautiful confirmation box.

5.4. Login Confirmation

To add a confirmation to the "Ok"-case of the login box, right-click in the corresponding `if`-part and choose *Add Fragment....* The *S60 Confirmation Note* is fine (see Figure 1), use "Login Successful!" as the label text. The difference to the query dialogue is that the note is created dynamically, whereas the structure of the login query is defined in the resource file.

Again, the generated code contains a huge portion of comments on how to use the confirmation note. In any case, you will have to include *aknnotewrappers.h* in *HelloWorldView.h*. The required library is already part of our project because of the query dialogue, so you don't have to add it anymore.

The two lines of active code from the fragment are the quick alternative for displaying a confirmation note. Again, you should prefer the advanced method of defining text in the *.loc* file as we did before. This tutorial is targeted at Symbian OS 7/8, so act according to the corresponding section of the explanation code. In short, it requires defining the following text in the *.loc*-file:

```
#define qtn_login_success "Login Successful!"
```

This text has to be converted to a descriptor (TBuf) resource in the *.rss*-file:

```
RESOURCE TBUF r_login_success { buf = qtn_login_success; }
```

Alternatively, you could right-click in the *.rss*-file, choose *Add Fragment...* and select *String Resource*. The comments will contain some more details on how to handle localized text.

To load the message from the resource file into the confirmation note, you have to add an additional library to the project (*CommonEngine.lib*) and include the header-file *StringLoader.h*. Now use the commented text for displaying the box instead of the simple method from above. If you are reading this text without trying it out, it might sound a bit strange. You'll notice what is going on when you work through this tutorial.

Now it's about time to write the *if*-statement for checking the user input. The *username* and *password* variables are initialized with default values, which you should probably make empty. After the user has successfully entered his user information, the variables will contain the text that the user wrote. So we simply have to compare the correct login text with the user input. The previously mentioned *Compare()*-function returns 0 if both descriptors are identical.

Once finished, create a Login Failure/Cancelled box. Refer to Listing 1 for the final code of `HandleLoginL()`. We're through with the most important part of the tutorial, congratulations!

Listing 1: The final login handling function

```
void CHelloWorldView::HandleLoginL() {
    _LIT(KCorrectLogin, "mopius");
    _LIT(KCorrectPwd, "asdf");

    // Show dialog 'Username: + Password:'
    TBuf<8> username(_L(""));
    TBuf<8> password(_L(""));
    CAknMultiLineDataQueryDialog* dialog =
        CAknMultiLineDataQueryDialog::NewL(username, password);
    if (dialog->ExecuteLD(R_LOGIN_QUERY)) {
        // Ok, index tells which item was selected
        if (username.Compare(KCorrectLogin) == 0 &&
            password.Compare(KCorrectPwd) == 0) {
            // Correct login
            CAknConfirmationNote* note = new (ELeave)
                CAknConfirmationNote();
            HBufC* notePrompt = StringLoader::LoadLC(R_LOGIN_SUCCESS);
            note->ExecuteLD(*notePrompt);
            CleanupStack::PopAndDestroy(notePrompt);
        } else {
            // Login failed
            CAknErrorNote* note = new (ELeave) CAknErrorNote();
            HBufC* notePrompt = StringLoader::LoadLC(R_LOGIN_FAILED);
            note->ExecuteLD(*notePrompt);
            CleanupStack::PopAndDestroy(notePrompt);
        }
    } else {
        // Login cancelled
        CAknWarningNote* note = new (ELeave) CAknWarningNote();
        HBufC* notePrompt = StringLoader::LoadLC(R_LOGIN_CANCELLED);
        note->ExecuteLD(*notePrompt);
        CleanupStack::PopAndDestroy(notePrompt);
    }
}
```

6. Importing an example

One of the most useful features of Carbide.vs is still waiting to be discovered by you. In my opinion it's one of the best ways to learn something if you take a look at existing examples and play around with them, modifying more and more fundamental parts. Today's Symbian OS / Series 60 SDKs come with more than 100 examples. Look for them in the *Examples-* and *Series60Ex-*directories.

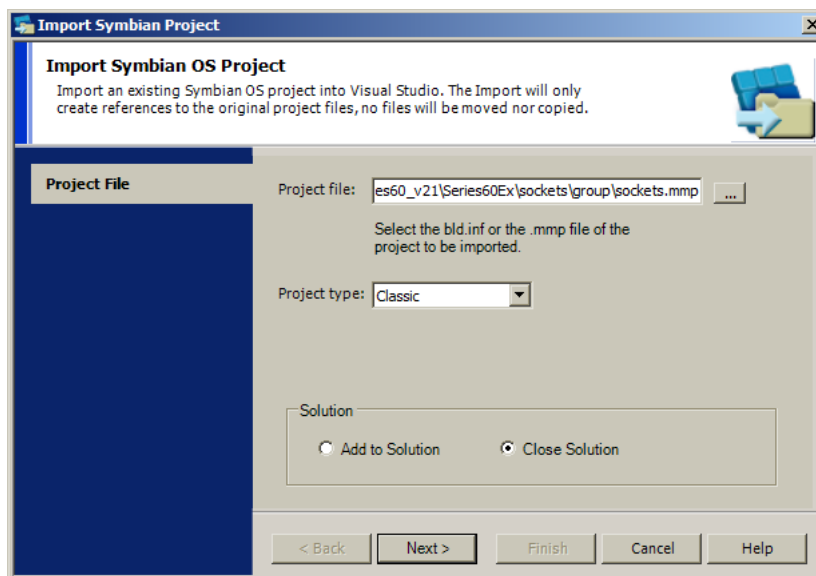


Figure 11: Countless sample applications are available that you just have to import

If you'd like to work with them from within the comfortable VS.net IDE, Carbide.vs provides a simple way of importing. Use *File->Import Symbian Project*, select the *.mmp*-file in the *group*-directory and in a few moments you will have the example completely integrated in a Visual Studio solution, without having to work with command line tools for compiling anymore (see Figure 11).

7. Conclusions

I do hope that your first steps with Carbide.vs and maybe even Symbian OS were successful and fun. With just a few clicks and some lines of code it was possible to write a fully functional mobile application featuring text input and comparison, bitmap display, a menu and a lot more.

If working through this tutorial was the very first time you have been developing for Symbian OS, many things might seem strange. In this case, it would be great if you are motivated to start searching for further answers on your way to get a professional Symbian OS developer. The references provided below are good sources for information, the book “Developing Series 60 Applications. A Guide for Symbian OS C++ Developers” from Leigh Edwards and Richard Barker is highly recommended if you want to expand your knowledge.

7.1. Additional Resources

- **Official Symbian Developer** site – documents and downloads
<http://www.symbian.com/developer/>
- **Symbian newsgroup** – the best place to get help
<nntp://publicnews.symbiandevnet.com/>
- **Forum Nokia** – loads of examples and downloads
<http://www.forum.nokia.com/>
- **NewLC** – news and developer information
<http://www.newlc.com/>
- **All About Symbian** – general news about Symbian OS phones
<http://www.allaboutsymbian.com/>
- **The Journey II** – a location based Symbian OS adventure game
<http://journey2.mopius.com/>

7.2.About the author

Andreas Jakl has founded a company for mobile application and game development called Mopius. He developed the location based Symbian OS games “The Journey I + II”, which managed to establish themselves as a reference for their genre. Recently, Andreas Jakl has done some courses dealing with development for mobile phones.

Email: andreas.jakl@mopius.com

Web: <http://www.mopius.com/>