RADFUSION, INC

Australian Devcon 2007

ABC Compliance Standards

RADFUSION, INC.

ABC Compliance Standards

© RADFusion, Inc.
PO Box 265
Dunedin, FL 34697-0265
Phone 727 738-9007 • Skype: radfusion

Table of Contents

Defining the ABC Standard 1
What are the ABC standards? 1
Coding the CLASS 3
  What about the CLASS declaration? 3
The Demo Class 3
Filling in the Class 5
Testing the CLASS 7
Coding ABC Compliant Templates 10
The Global Extension 10
Setting the default instance 11
Preprocessing the default instance 12
Exporting the class 12
Adding developer dialogs 13
Changing the Base Class 15
Getting the Class Names 15
Generation of the Instance 16
Make an Embed Point 16
Embeds for Public Methods 17
Generate the Instance 18
Global Embed Points 19
Testing the Template 20
Local Template Wrappers 25
Local Class Dialog 25
Setting Default Instance Names 25
What about Embed Points? 25
What Was Not Discussed 26
Useful Things 26

Chapter 1

Defining the ABC Standard

In order to make ABC Compliant code, you need to know what the standard is

To define the term, *standard* means "something that is adequate for a particular purpose." Many Clarion developers code classes, some of them code templates for their classes. Not many understand how to write a class with a template wrapper to the ABC standard. I can't really blame these developers as this standard has not really been broadly disseminated in an easy to understand fashion.

Templates themselves can sometimes feel like you are aiming at moving targets, so hitting it is not easy. Today's talk is not about the nuances of template coding and there are many nuances. Instead, I'm focusing on just what is needed to make a CLASS and its template wrapper ABC compliant.

ICON KEY

Valuable information

Test your knowledge

Keyboard exercise

Workbook review

To start this, let's list what features are needed.

What are the ABC standards?

In order for any CLASS, no matter how simple or complex could be considered ABC Compliant, it must meet these criteria:

- If the class is used in an application, the classes are included automatically. As a corollary, if the class is not used, it is not linked in; again automatically. This must be done without extra templates that include or exclude a given class. Classes must cater to hand coders too and the ABC classes shipped by Softvelocity are quite suitable for hand coders.

- The class used in any application must work unchanged by the developer. This means it *must* work correctly out of the box in all phases. No changes are required by the developer for it to work.

- The class must allow changing if the developer wishes to alter the label and it still must work. Many developers work in environments where naming standards rule the day, or in the case of multiple instances of the same class, meaningful names from the default names are allowed.

- The template must allow embeds for all public methods, not just virtual methods. It must also color code these embed points as to their type.

- If a code embed is needed by the developer for whatever reason, the template must generate correct code, with the parent call if applicable. In such cases, the proper class generation and instantiation of the class happens on behalf of the developer. This is also true even when the developer does not use an embed point for a class.

- The template must allow for extension of a given class. This means the developer can add new properties and methods without leaving the application itself and more importantly, without editing the original code. These new methods and properties will also have their own embed points, correctly color coded.

- It must allow for proper DLL use, if required. This means exporting the class and its methods for use in other applications if needed. This varies depending on the template you write. A developer should never have to use a template to include or exclude any ABC compliant class.

- If applicable and required, the base class in use could be changed. This means you could completely replace an ABC class with one of your own, or use a new class that has an ABC class as its parent. This is the easiest use since it does not require a template, even inside application files.

Those are the specifications that ABC compliant class code must meet. There have been two *Clarion Magazine* articles on this; it is a section in my book *Programming Objects in Clarion* and a subject of various educational classes lead by various instructors including yours truly.

Chapter

2

Coding the CLASS

What about the CLASS declaration?

This is the easy part. Code your CLASS as you normally would. There really is nothing different than you normally do with some small exceptions that really don't alter the compliance issue. Those are discussed later.

For illustration purposes, I've provided an extremely simple CLASS. It really won't do anything useful except some visual feedback to show the methods executing.

For this talk, the CLASS uses two source files, the INC file, which is the CLASS definition itself, the other a CLW which is the code for the methods in the CLASS. Keep in mind that you don't have to code CLASS definitions this way; it has no bearing on ABC compliance. I'm using this style as it is quite common.

The location of this source file is important. In Clarion 6, these file are located in *%ROOT%\libsrc* where *%ROOT%* is the install location of Clarion 6. In Clarion 7, it is different. The folder is *%ROOT%\libsrc\win*. The distinction here is important as the other libsrc folder is for Clarion.NET. Reason being is the Clarion 7 IDE supports both Clarion (as in Win32 applications) and Clarion.NET.

The Demo Class

The CLASS declaration itself is nothing really special. A CLASS is a CLASS, or is it? The first line of the include file should have this little comment:

```
!ABCIncludeFile
```

I'm sure everyone has seen this before. But what does it do? This is just a "tag" for the templates to pick up on. However, I've left this comment incomplete. This comment takes a parameter and if left off, it defaults to "ABC" which is the same as coding the comment like this:

```
!ABCIncludeFile(ABC)
```

You really don't want to do this as the templates that are to follow will assume that this is an ABC class and the point of this talk is not how to code ABC classes. I'm showing

you how to code ABC *compliant* classes. It is worth mentioning that you do want the above style if you are coding classes that are so key to the running of your application, they must *always* be included. ABC has classes like this for example the ***ErrorClass*** and ***FileManager*** classes. Since this is a demo, let's make the comment like this:

```
!ABCIncludeFile(DEMO)
```

The parameter is known as a family classification. You can have many. The point is you *don't* want ABC as the family classification. If you do, then the classes declared as belonging to the ABC family are linked into your project even if you don't use them. That is not really a bad thing; you just have dead code in your binaries. The good part of this is if you do this, they should be linked correctly even when making a DLL. But let's stick to the ABC standard. If you don't use this demo class in your project, it's not linked.

```
!ABCIncludeFile(DEMO)

OMIT('***')
 * Created by Clarion 7.0
 * User: RADFusion, Inc.
 * Date: 4/1/2007
 * Time: 2:21 PM
 *
 * To change this template use Tools | Options | Coding | Edit Standard Headers.
 ***

DemoClass     CLASS,TYPE,MODULE('Devcon2007.clw'),LINK('Devcon2007.clw',_DEMOLinkMode_),DLL('_DEMODLLMode_)
              END
```

Figure 1 - The Start of the DemoClass as seen in the Clarion7 editor

This is the beginning of the CLASS itself. At the moment, it has no properties, no methods. Take note of the attributes. CLASS is obvious. The TYPE attribute says this declaration is just that; a declaration. Without that attribute, it would be a declaration and an instance. The template wrapper takes care of instantiation so you don't need to be concerned with it.

The MODULE indicates where the compiler can find the code for the methods. The LINK attribute should be of interest here. This means "link in the source module shown here without having to add it to the project list." The second parameter is a project define that is either true or false. If it is false, then it would act as if the MODULE attribute was left off the declaration.

The DLL attribute declares a variable, FILE, QUEUE, GROUP, or CLASS defined externally in a .DLL. The parameter or flag if set to false tells the compiler to treat the declaration as if the DLL attribute is not present, any other value is true and indicates the class is declared elsewhere.

That is all you need for the declaration. The upcoming template wrapper will take care of the rest.

Filling in the Class

A CLASS should do something or at least provide a base where something meaningful could be done when used in context. My base classes by themselves don't usually do anything useful since they are meant to be derived in an application. That concept is not really the focus of this talk. Classes may be specific or vague, abstract if you will and classes designed like this are still subject to the ABC standard, perhaps even more so.

Today, I'm keeping this simple since this is a teaching presentation. This CLASS as I mentioned previously will only have a few methods with some visual feedback to show they are working. One useful way is with the Clarion MESSAGE statement. I need to add a *ShowMessage* method as seen below:

```
ShowMessage       PROCEDURE(),VIRTUAL
```

Keep in mind this is merely the declaration of this method, the code for this method comes later. As the name implies, this method will show a message to the user. I've also added the VIRTUAL attribute. I added it only to set up the template embeds that are coming later as there is a special treatment for VIRTUAL methods.

I'll add some other methods and properties at this time so the complete class looks like this:

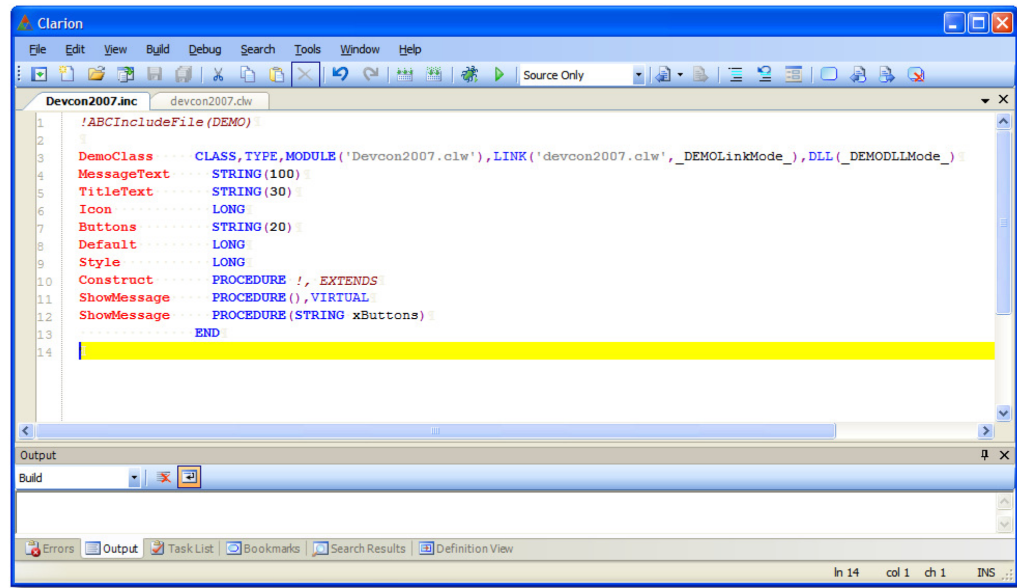

Figure 2 - More of the DemoClass

Notice the *Construct* method. It has a special comment and this comment has a special meaning which I'll explain in the template section.

In the module code, there needs to be some code for the methods:

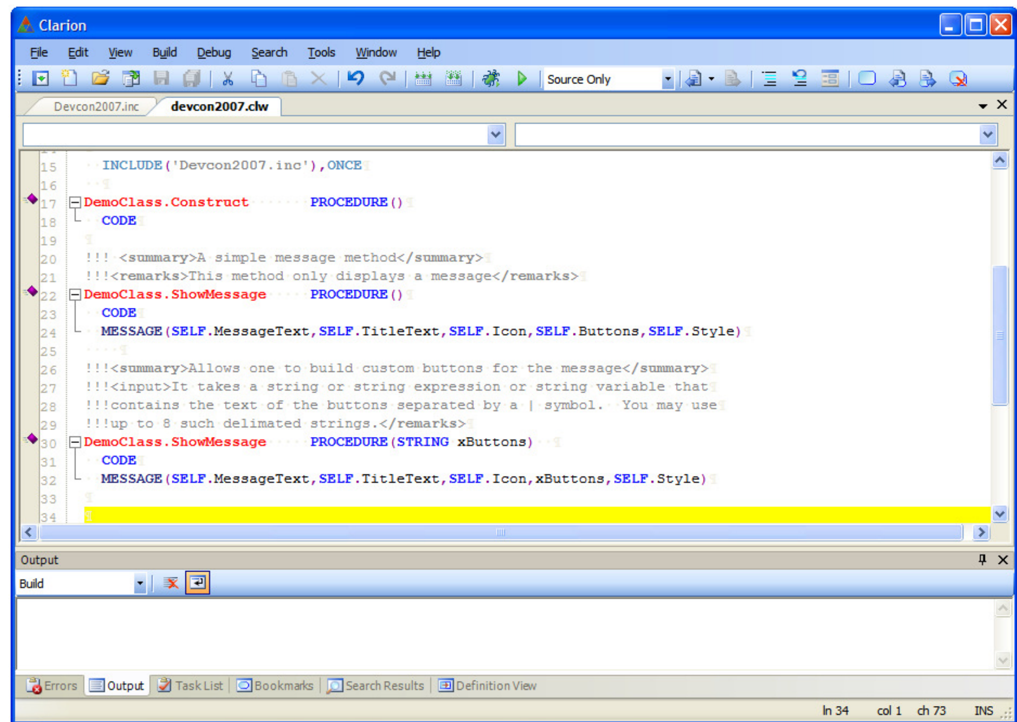

Figure 3 - Module code

Just a special note; there seems to be an excessive amount of comments. This taking advantage of Clarion 7 features, but everything you see here is legal in Clarion 6. These features are not required or even needed for ABC compliance; yet they could be helpful when you are writing code. Those comments assist you when you write code using those methods. Clarion 7 has intellisense and these comments appear in the tooltip. In the main body of code, only the prototypes are seen in the tooltip.

Testing the CLASS

In order to test this small class, it requires a test project. This is a good idea to ensure your class works as intended before you start the template side of things. The following code is a good test:

```
1
2    PROGRAM
3
4    OMIT('***')...
12
13   MAP
14   END
15
16   INCLUDE('EQUATES.CLW')
17   INCLUDE('devcon2007.inc'),ONCE
18
19 DC  DemoClass
20
21   CODE
22   DC.Buttons = '&Chocolate|&Vanilla|Stra&wberry|'
23   DC.DefaultButton = 3
24   DC.Icon = ICON:Question
25   DC.MessageText = 'What is your favorite flavor?'
26   DC.Style = 0
27   DC.TitleText = 'Pick a flavor...'
28   DC.ShowMessage()
29
```

Figure 4 - The test code

There is one thing you must do at this stage, especially if you are a hand coder. The class as written won't work with a new project using only defaults. If you do, you get this result:

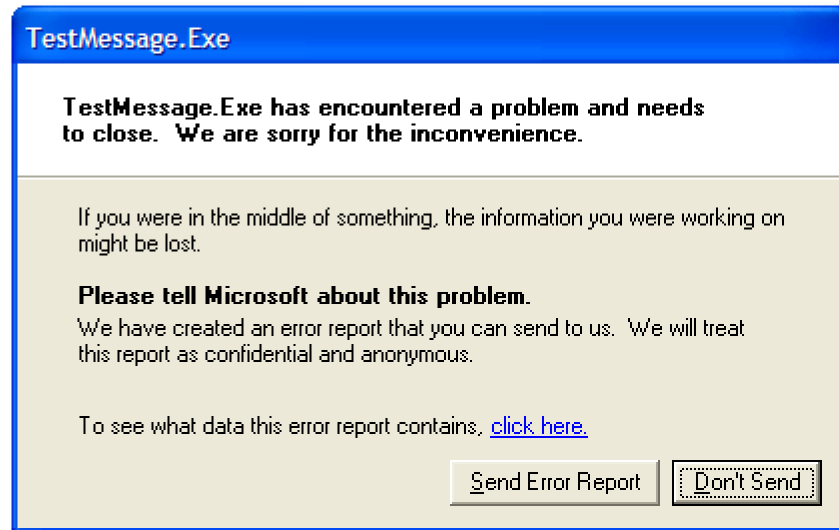

Figure 5 - The code crashes.

Recall the attributes of the CLASS, especially the LINK and DLL attributes. It uses flag parameters and since a new project does not have these variables set or defined, the result is trying to instantiate a CLASS that cannot be instantiated.

The test code crashes on this line:

```
DC    DemoClass
```

Add these flags and it won't crash:

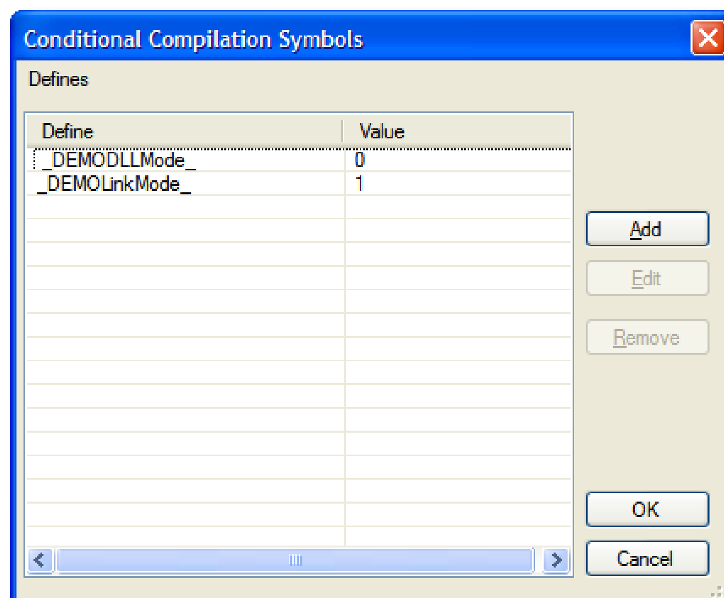

Figure 6 - Don't forget the flags!

Executing this program shows the following:

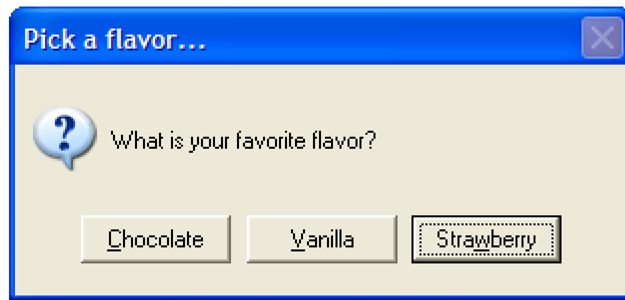

Figure 7 - The executing program.

This proves the code and its underlying CLASS does something. Now it's time to make the template.

Keep in mind that I did not actually write any code in the *Construct* method. It also has a special comment. I plan to let the template take advantage of this by treating this method as a virtual method.

Chapter

3

Coding ABC Compliant Templates

Now there is a CLASS in *libsrc*. As it sits right now, it is ready to go if you are a hand coder. This is fine if you wish to use the CLASS in your project. The hand coder is responsible for all aspects of the code from this point onward, not templates.

Template developers and users let the templates do the work, thus freeing the developer to concentrate on the features of their application. This is where the real magic happens as it means you drop in the template and it works. By that I mean the correct generation of the code that uses the CLASS.

The first line of code in your new template file is as follows:

```
#TEMPLATE(MessageClass,'MessageClass Template'),FAMILY('ABC')
```

All template files (usually with a TPL extension) begin with the #TEMPLATE statement. Add the required template name and description text then the FAMILY attribute. For coding ABC compliant template wrappers, this is required since the default attribute if you leave it off is "Clarion", referred to as "legacy". You can add both families if you want since legacy applications can benefit from objects too.

The Global Extension

Typically, ABC compliant templates tend to be extension templates, but you are not restricted to this usage. For this talk, I'm using an extension. Since this class could be used anywhere in an application, I've made this a global extension. This means you can add it only via the global dialogs in an application. This means my next few lines of template code appear as follows:

```
#EXTENSION(DemoClass,'Global Demo Message Class'),APPLICATION
#BOXED('Default prompts'),AT(0,0),WHERE(%False),HIDE
  #INSERT(%OOPHiddenPrompts(ABC))
#ENDBOXED
```

The APPLICATION attribute means this extension is available only via the global dialogs. If I wanted it available in a procedure or local level, then the attribute would

have been PROCEDURE instead. This would be proper if you could have more than one instance of a CLASS, such as a class for a browse list for example.

Notice the #BOXED statement. This is just a template statement that groups things in a visible box. It takes the same AT attributes as any window or control. In this case, both the top and the left side start at position 0. This is fine as it never displays. Notice the WHERE attribute. This specifies the box is visible only when its expression is true. The HIDE attribute specifies the prompts are hidden if the WHERE expression is false when the dialog is first displayed. If no WHERE attribute is present, prompts are always hidden. I could have left off the WHERE statement, but I think this improves readability.

The #BOXED structure has only one line of code in it, #INSERT. This inserts a piece of template code where #INSERT is. What it is really doing is inserting a special block of template code called a #GROUP. This block of code actually lives in the ABC templates. You can see this by the family attribute of the #GROUP name. #GROUPs have names and template variables or names always start with the percent sign. In this case the name is %OOPHiddenPrompts. As the name suggests, it is not meant to be visible. This is one way to do some behind the scenes processing without a need to display everything and messing up your dialogs.

If you want to save some lines of template code, you could easily code this:

```
#INSERT(%OOPPrompts(ABC))
```

If you ever want to see this template code, just open *ABOOP.TPW*, as most of the #GROUPs you need to call live there (as of Clarion 6).

So what does it do? It keeps a list of all the known classes and their properties and methods!

Setting the default instance

The next thing you need to do is ensure that all the classes are loaded, and if not, then load them. This is done inside the #PREPARE statement:

```
#PREPARE    #!Set class name in case developers never edit it
  #CALL(%ReadABCFiles(ABC))    #!Read ABC class headers if needed
  #!Set local name from libsrc class name
  #CALL(%SetClassDefaults(ABC),'DC','DC','DemoClass')
#ENDPREPARE                              #! end #PREPARE
```

Inside the #PREPARE structure there are two calls to ABC #GROUPs; %ReadABCFiles and %SetClassDefaults. %ReadABCFiles determines if there is any classes loaded internally and if not, loads them. This is the template code responsible for the dialog you see when Clarion is loading the ABC header files. This is done when you open your first application since launching the Clarion IDE or your **press** the READ ABC HEADER FILES under the *Class* tab in the global properties of an application.

The other #GROUP sets an instance name for the class. Remember, this demo class uses the TYPE attribute, so having the templates name an instance is a useful thing to do. It takes 3 parameters, the *tag*, *object name* and *object type*. This #GROUP then calls other internal #GROUPS to get things set up. The tag and object name are usually the same in a global instance. Where they would be different is local instances inside a procedure. For example a tag of *BRW* and an object name of *BRW1*. The *object type* parameter refers to the class label itself, which is a TYPE.

A #PREPARE statement executes first, before any other template code. Think of it like a template equivalent of setup. This is what makes a class work out of the box. This satisfies one of the points of ABC compliance.

Preprocessing the default instance

This is just like the #PREPARE structure. This is really an exact repeat as above with the following difference:

```
#ATSTART    #!Execute this code before other template statements
  #CALL(%ReadABCFiles(ABC))    #!Read ABC class headers if needed
  #!Set local name from libsrc class name
  #CALL(%SetClassDefaults(ABC),'DC','DC','DemoClass')
#ENDAT                              #! end #ATSTART
```

#ATSTART processes all code in its structure before the rest of the template. It should never be used where you generate code as this structure is really for setting up the rest of the template code. A good example is initializing template symbols to known values before they are even displayed in dialogs.

In this context, if a developer changed the name of an instance of a class, this structure ensures the name is remembered before any dialog displays. In other words the #PREPARE statement gets the default name, the #ATSTART gets the name (changed or not) before the developer gets access to it. This ensures that no matter what, there is a valid instance name ready for use.

Exporting the class

Chances are that if a class can live across DLLs (like the ABC classes), then you want them exportable. If anything is not exported, it means that it is for the exclusive use of the current application file. Outside access is not possible. While there may be times this is what is needed, more often than not, you do want one instance of a class available to all applications in a solution.

The templates support this use and here is the code required to do it:

```
#AT(%BeforeGenerateApplication)       #!For exporting the class
#!The parameter in the !ABCIncludeFile comment in the INC file
  #CALL(%AddCategory(ABC),'DEMO')
#!Used for DLLMode_ and LinkMode_ pragmas (see defines tab
#!in project editor)
  #CALL(%SetCategoryLocation(ABC),'DEMO','DemoClass')
#ENDAT             #! end #AT(%BeforeGenerateApplication)
```

The #AT statement means "at this embed point", a useful statement for embed points that are for the use of template code (although not always for that purpose). This means that all code inside of it generates at this location as indicated in the #AT parameter. The comments inside this code describe what the template code following the comment does. The first #CALL processes all classes with "DEMO" in the comment parameter. This is must be in upper case.

The next #CALL is what makes _DEMODLLMode_ and _DEMOLinkMode_ flags and sets their values. It is what gives them a value in your global application and a different set of values in the applications that follow (if you use it).

This satisfies the need to export the class depending on how you build it.

Adding developer dialogs

Up until now, nothing is visible to the developer. One of the standards is that the template wrappers must allow the developer to change the default name of the class if he so wishes. This is a very good idea if you could have more than one instance of a class in a procedure.

For example in the shipping ABC templates you could have more than one browse list on a window. The default names are fairly meaningless and often confusing. BRW3 does not tell you exactly which browse control is the 3rd instance of a browse class. And if you wish to add embed code for this, how can you be sure you are dealing with the Customer Order list box?

Such naming conventions are fine since that satisfies the "must work out of the box" requirement. But being forced into those naming conventions could be problematic. Thus there is a provision for a developer dialog allowing changes to the name of any instance of a class. It must still generate correct object code, regardless what the developer does.

The following code does just that:

```
#TAB('Global Message &Instance')  #!Global object dialog
  #BUTTON('&Message Class')        #!Display a button
    #WITH(%ClassItem,'DC')         #!Show the global instance name
      #INSERT(%GlobalClassPrompts(ABC)) #! global prompt dialog.
    #ENDWITH                       #! end #WITH(%ClassItem,'DC')
  #ENDBUTTON                      #! end #BUTTON('&Message Class')
#ENDTAB                   #! end #TAB('Global Message &Instance')
```

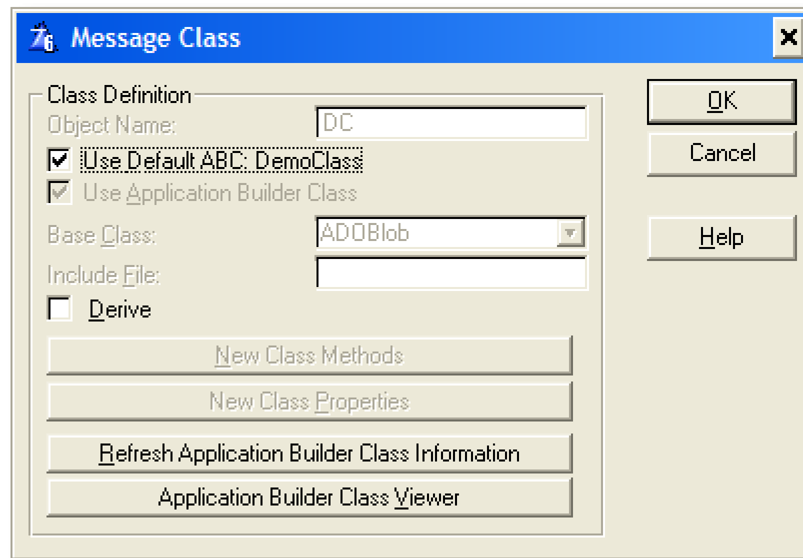

Figure 8 - The resulting template dialog

The above template displays the dialog as shown in Figure 8. Notice that the title came from the #BUTTON text. This dialog should be familiar to all ABC developers. Since this is a global dialog, changing of the object name is not allowed. There are two reasons for this:

1) Since the object is global, there should be only one instance.

2) There is a template symbol that gives you the instance count and the value is always null in a global scope. You should use this in a local instance.

There is something else about this dialog; you can add new methods and properties to the class, all without touching the original source. This satisfies yet another ABC compliant standard.

If you do build local class dialogs, then use %ClassPrompts in the above template code instead. This allows you to change the object name, for example changing *BRW1* to *CustomerList*.

Changing the Base Class

Another standard of ABC compliance is to allow a developer choose another class instead of the default. This is purely an option and the developer is under no obligation to change the class. If you coded a class that you like better than say ABC's *WindowManager* or even a child class of *WindowManager*, then it would make sense to substitute the class the templates base their class code on.

Here is the template code you need for this dialog:

```
#TAB('Message &Base Class')         #!Global class dialog
  #PROMPT('&Default class:',FROM(%pClassName)),%ClassName,|
           DEFAULT('MessageClass'),REQ
  #DISPLAY()
  #BOXED(' Usage ')
    #DISPLAY('If you have another class you wish to use |
             `instead, select it from the list or use the |
             `default shown.')
  #ENDBOXED
#ENDTAB                          #! end #TAB('Message Base Class')
```

Note: the above code is wrapped for readability. Refer to the actual template code for actual appearance.

This dialog meets the ABC standard that a developer is not locked in to a particular class and is free to change it.

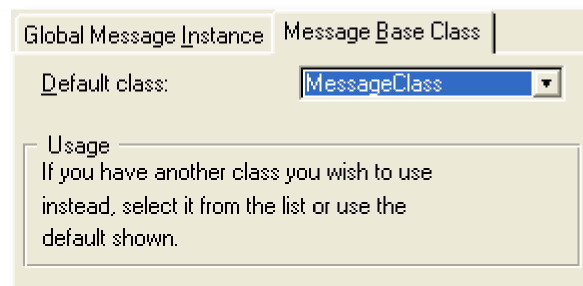

Figure 9 - The default base class dialog.

The main advantage here is that if you do wish to replace an ABC class, then you don't even need a template to do so. There are many such dialogs for all the major ABC classes. The drop down list shows all the known classes and you may pick from any of them, if it applies.

Getting the Class Names

Ever wonder how the ABC templates get the class names? Reading the header files is part of the process, but this usually means that the instance name is unknown.

The next set of template code appears as follows:

```
#AT(%GatherObjects)           #!Ensure objects are known and loaded
  #CALL(%ReadABCFiles(ABC))     #!Read ABC class headers if needed
  #CALL(%AddObjectList(ABC),'DC')     #!Add the template object
  #ADD(%ObjectList,%ThisObjectName)   #!Add the object to the list
  #SET(%ObjectListType,'DemoClass')   #!Set the base class name
#ENDAT                                #! end #AT(%GatherObjects)
```

One thing to notice about this section of template code; you don't need to #DECLARE the symbols shown. The ABC templates have already done this and since you are calling ABC template code, they are known. If you are really curious, you can inspect the template code in the ABC templates to see what they are doing. But what is happening here is that the instance is matched to the class, and thus making it available to the rest of the templates, even those you did not code.

Generation of the Instance

At this point, the template knows about the instance and the class. When you generate code, the templates must correctly generate code, even if the developer knows nothing about coding classes. The following code covers one part of this (wrapped for readability):

```
#AT(%GlobalDataClasses)               #!At global class embed point
  #CALL(%SetClassItem(ABC), 'DC')     #!Set the current instance
  #INSERT(%GenerateClass(ABC), 'DC','Global instance and
definition'),NOINDENT
#ENDAT                              #! end #AT(%GloballDataClasses)
```

This small but vital template section is responsible for generating the correct instance in your application.

Make an Embed Point

The following template code generates the embed point for all public methods in your class, including the "parent" call (again, wrapped for readability).

```
#AT(%DemoClassMethodCodeSection),PRIORITY(5000),|
    DESCRIPTION('Parent Call'),|
    WHERE(%ParentCallValid())   #!Add parent call embed point
  #CALL(%GenerateParentCall(ABC))    #!Generate the parent call
#ENDAT                  #! end #AT(%DemoClassMethodCodeSection)
```

This is another embed point, and the point is always the expression of the *class name* + "*MethodCodeSection*." Parent calls are always priority 5000 since there are a potential 10,000 embed points in any given method. The description attribute is the string you see in the embed tree. Notice the WHERE attribute. This is a filter that evaluates as either true or false. You cannot place #AT structures inside of #IF structures, the template language won't allow it.

What is interesting about the WHERE attribute is the expression in it. It looks like its calling a function and you would be correct. In the template language a #GROUP can act like a function call, even returning values. However, there is a small catch. The %ParentCallValid() is an ABC #GROUP. But you cannot use the "ABC" parameter in this case since it will think you are passing a parameter. So how does one call an ABC #GROUP without it?

Try this bit of code and see if you can see what is going on:

```
#GROUP(%ParentCallValid),AUTO
#DECLARE(%RVal)
#CALL(%ParentCallValid(ABC)),%RVal
#RETURN(%RVal)
```

The "trick" is to make your own #GROUP, which in turn calls the ABC #GROUP as expected. In this case, there is a #DECLAREd symbol which stores any return values, which this local #GROUP then returns to the caller, in this case the WHERE attribute of the #AT. The AUTO attribute means any symbols declared in the #GROUP are declared only for the caller and no one else. This means you can't access any symbols there from a #PROCEDURE template, you need to #DECLARE those symbols in another scope.

Embeds for Public Methods

This next bit of template code is the biggest shown so far. Its purpose is to generate embed points for all public methods in a class. I've wrapped it for readability as a few of these lines are quite long.

```
#IF(%BaseClassToUse())          #!If there is a base class
  #CALL(%FixClassName(ABC),%BaseClassToUse())
  #!Assign the base class, cleaning up any errors
  #FOR(%pClassMethod)          #!For every method in this class
    #FOR(%pClassMethodPrototype),WHERE(%MethodEmbedPointValid())
    #!and the prototype is not private
      #CALL(%SetupMethodCheck(ABC))
      #!ensure the proper instance, any overrides generate
      #EMBED(%DemoClassMethodDataSection,|
             'DemoClass Method Data Section'),|
              %pClassMethod,|
              %pClassMethodPrototype,|
              LABEL,DATA,|
              PREPARE(,%FixClassName|
             (%FixBaseClassToUse('DemoClass'))),|
             TREE(%GetEmbedTreeDesc('DEMO','DATA'))
      #?CODE           #<!Add CODE statement for method
      #EMBED(%DemoClassMethodCodeSection,|
             'DemoClass Method Code Section'),|
             %pClassMethod,|
             %pClassMethodPrototype,|
             PREPARE(,%FixClassName|
             (%FixBaseClassToUse('DemoClass'))),|
```

```
              TREE(%GetEmbedTreeDesc('DEMO','CODE'))
      #CALL(%CheckAddMethodPrototype(ABC),%ClassLines)
      #!Generate the prototype and structure for each method
    #ENDFOR         #! end #FOR(%pClassMethodPrototype),
  #ENDFOR           #! end #FOR(%pClassMethod)
  #CALL(%GenerateNewLocalMethods(ABC),'DEMO',%True)
  #!Generate any new methods code from class dialog, if present
#ENDIF              #! end #IF(%BaseClassToUse())
```

That is a hefty chuck of template code! What does it do? To summarize, it builds an embed point for each method and an embed point so you could declare data inside that method. This is what generates the data and code embeds for each public method in your class. As you can see it is doing a lot of work to get this to happen, mostly lookups inside the template's own queue system, filtering out the methods that don't belong to a given class, and then ensuring these methods are entitled to a data and code embed.

Generate the Instance

Up until this point, there has been little to see. Everything up until now is getting things set. Generating an instance is a must. If you don't and you somehow manage to get past the compiler, your application will produce a GPF upon startup. The following lines of code handle the instance generation:

```
#AT(%GlobalData)
  #INSERT(%GenerateClass(ABC),'DC'),NOINDENT
#ENDAT
```

In the global data of this application, this template code generates the correct instance of this class. It appears as follows (with other ABC global classes):

```
DC                     DemoClass

FuzzyMatcher           FuzzyClass     ! Global fuzzy matcher
GlobalErrorStatus      ErrorStatusClass,THREAD
GlobalErrors           ErrorClass     ! Global error manager
INIMgr                 INIClass       !non-volatile storage manager
GlobalRequest          BYTE(0),THREAD
GlobalResponse         BYTE(0),THREAD
VCRRequest             LONG(0),THREAD

Dictionary             CLASS,THREAD
Construct                PROCEDURE
Destruct                 PROCEDURE
                       END
```

The yellow highlighted line of code above is the class. It generated an instance of *DemoClass*. Also notice that I've used the NOIDENT attribute. I like using indentation in my template code as for me it makes it easier to read. However, the rules state that using #INSERT in this fashion will generate the code at the exact spot

the #INSERT appears. In other words, it appears in column two. This is an error when you are declaring data since column one is reserved for declaring data entities.

There is also another benefit this code produces. If your application does not know about your INC file, then how can it compile clean? It cannot. The code generates the proper INCLUDE statement as well.

Global Embed Points

No template discussion would be complete without talking about how to generate embed points in the embed tree. In order to bring this off, you need this bit of template code:

```
#AT(%ProgramProcedures), WHERE(%ProgramExtension<>'DLL' |
    OR ~%GlobalExternal)
  #CALL(%GenerateVirtuals(ABC), 'DC',|
        'Global Objects|Message Template',|
       '%GlobalEmbedVirtuals(MessageClass)', %TRUE)
#ENDAT
```

This template code states that if this application is an EXE, then produce the embed points in the embed tree. Otherwise you won't find any embed points. If you need embeds in each app, then another approach is needed and I'll discuss local embeds shortly.

If you inspect this code closely, you see it calls a local group. "MessageClass" is the name of this template. Therefore, the %GlobalEmbedVirtuals is a #GROUP that lives only in this template. Its code is as follows and should look familiar:

```
#GROUP(%GlobalEmbedVirtuals, %TreeText, %DataText, %CodeText)
#EMBED(%MessageClassDataSection,|
       'MessageClass Method Data Section'),|
       %ApplicationTemplateInstance,|
       %pClassMethod,%pClassMethodPrototype,|
       TREE(%TreeText & %DataText)
  #?CODE
  #EMBED(%MessageClassCodeSection,|
         'MessageClass Method Code Section'),|
         %ApplicationTemplateInstance,|
         %pClassMethod,%pClassMethodPrototype,|
         TREE(%TreeText & %CodeText)
```

So what does this do? It produces embed points like the following screen shot:

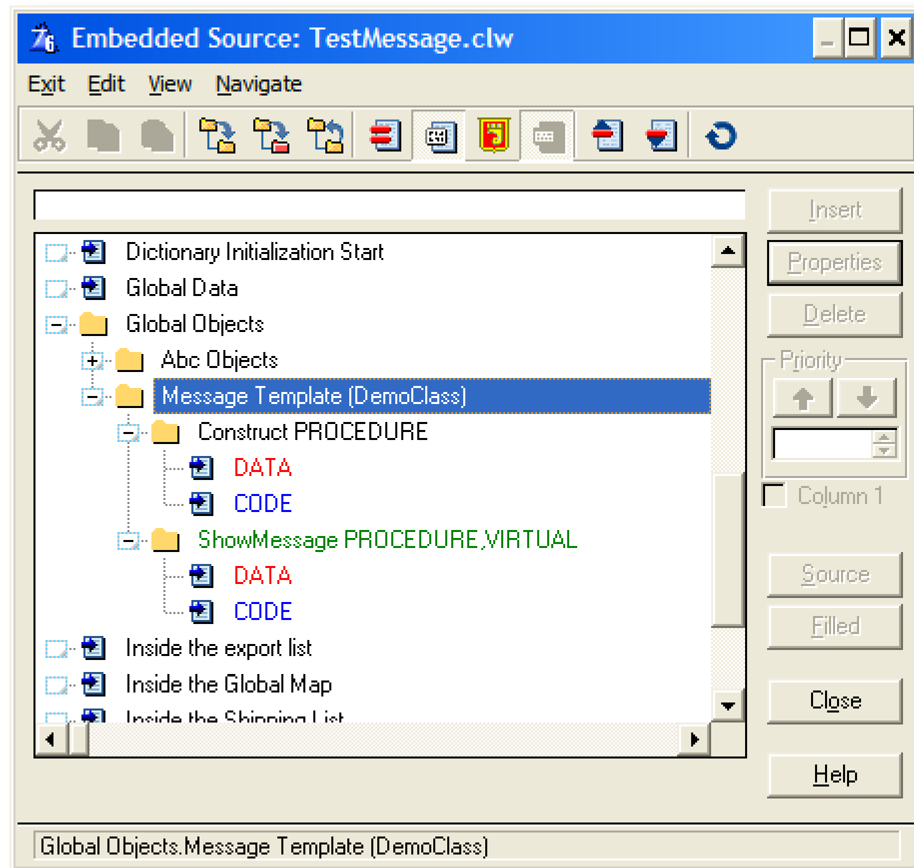

Figure 10 - The global embed tree.

This tree shows all the public methods in the class. You can see the *ShowMessage* virtual method is in the correct color and shows its data and code embeds. The *Construct* method is not a virtual method but it is displayed anyway, which I'm about to explain why. The other *ShowMessage* method is not visible at all, despite being non-private. What gives?

If you recall when I discussed the class definition, the construct method had a special comment on it. That was "!, EXTENDS". This means non-virtual public methods do appear in the tree. There is another special comment, rarely used. Suppose you want to hide a virtual method from appearing in the embed tree? Add the comment, "!, FINAL" and it will vanish.

Testing the Template

What happens if I add some source to an embed point like this:

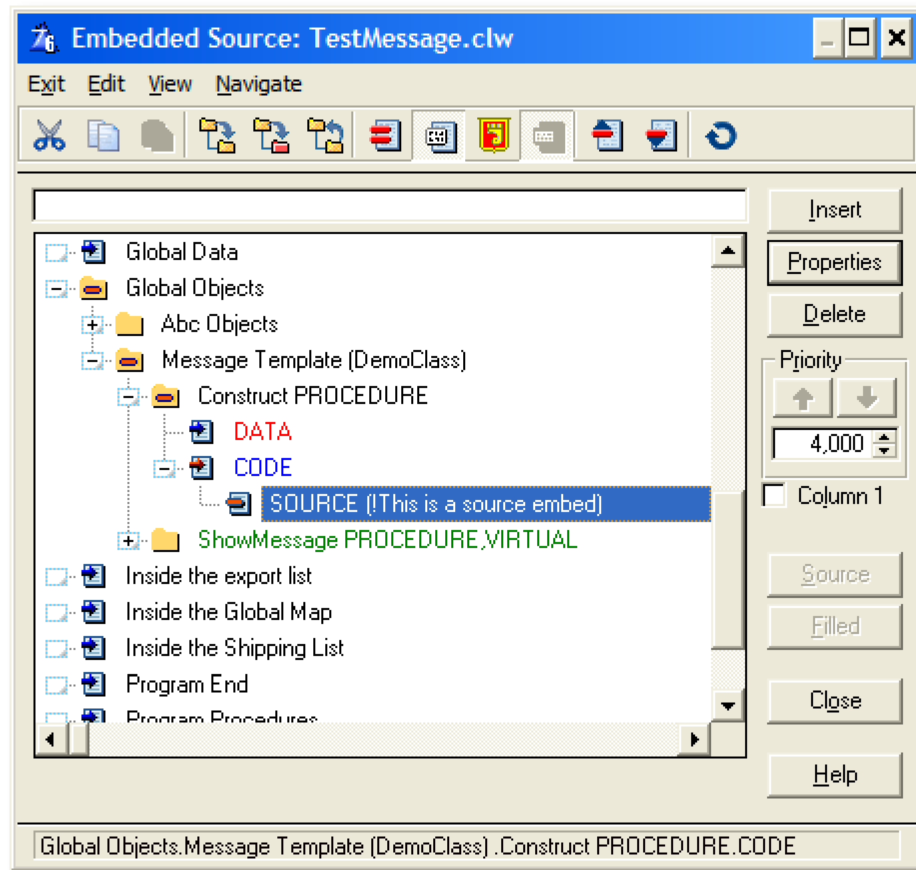

Figure 11 - Filled in embed point.

What happens when I generate source? For one thing, the generated instance will look a bit different:

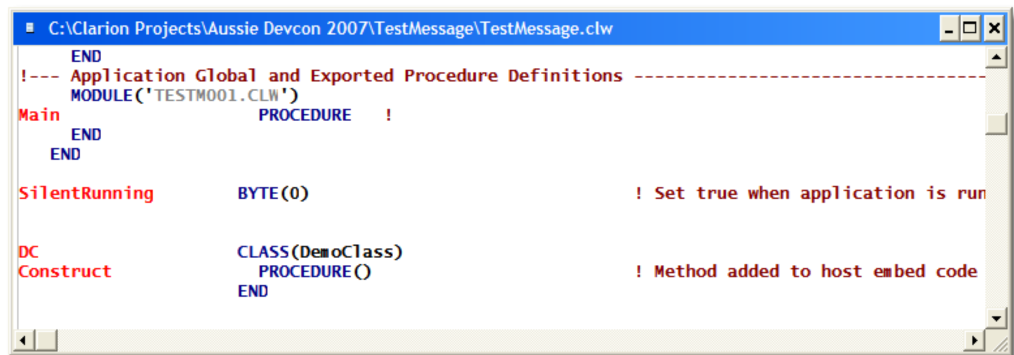

Figure 12 - Class declaration changed.

Instead of the simple instance declaration, the templates saw the developer is overriding a method, does not care which one. For each embed used, the template generates the method name and prototype. You can still see it's an instance of

DemoClass. That is fine for the class declaration, what about the embed itself? Scrolling down one sees the method override and the embed source:

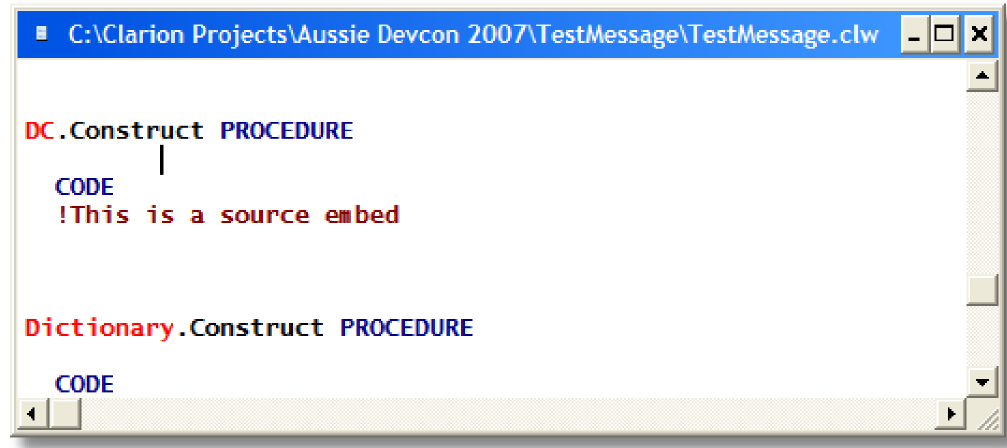

Figure 13 - Embed in method code.

It is also generated in the proper location. You can see that a template wrapper can remove the tedium of coding in OOP style. But will it work? There is one other spot to inspect to ensure that running a test app won't GPF instantly and that is in the project system.

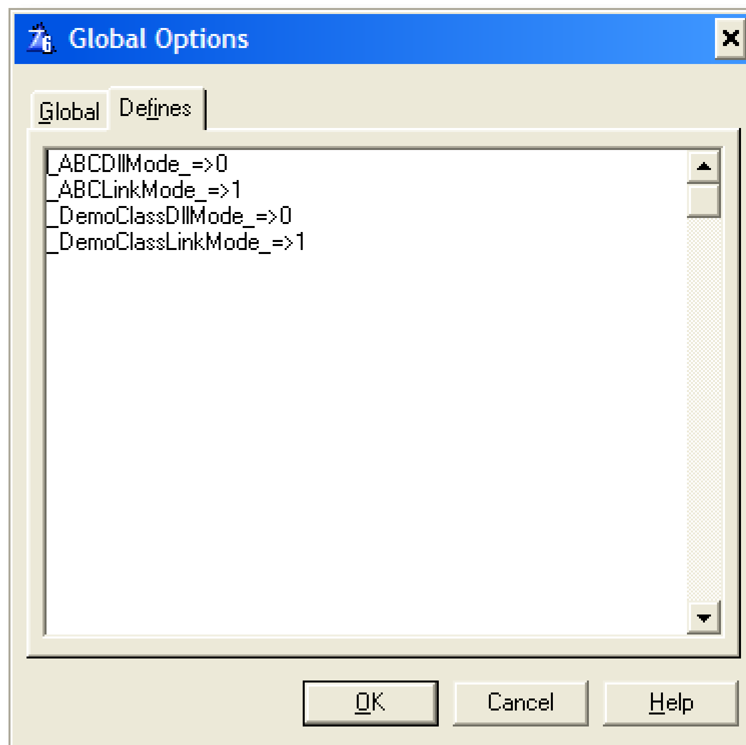

Figure 14 - Project defines generated by the template.

This is the same defines listed in the *DemoClass* and generated by the **%AddCategory** and **%SetCategoryLocation** template code discussed prior. More importantly, this gives you a way to ensure that if the template is not used in an application, the classes are not linked in. And more importantly, it won't break existing apps which you would risk doing if you stuck with the ABC defines instead of using your own. This also means you don't need to ship templates to include or exclude this class.

What about extending the class without touching the code in libsrc? The idea behind this is that extending a class is unique for the current application. You do this via the global class dialog.

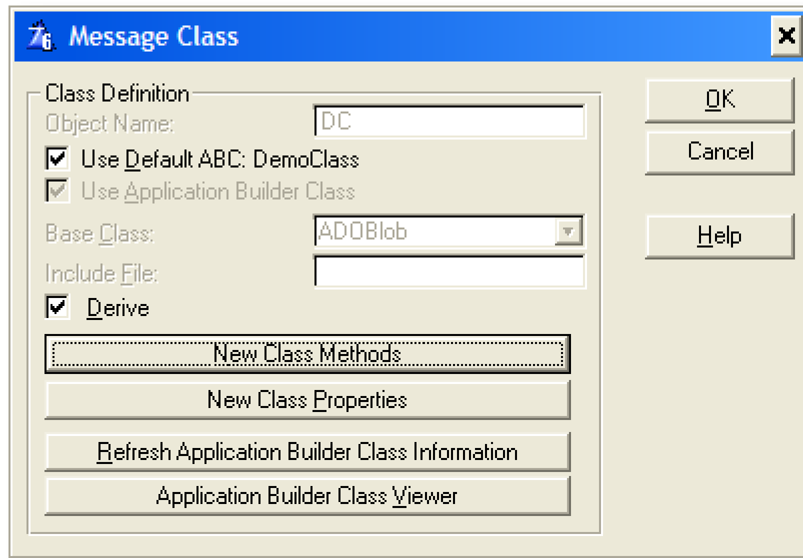

Figure 15 - Deriving the class.

Once you set the DERIVE check, the NEW CLASS METHODS and NEW CLASS PROPERTIES enable. Each gets their own dialog. You may refer to the Clarion help system for a full description of how this works.

What does the generated class look like now?

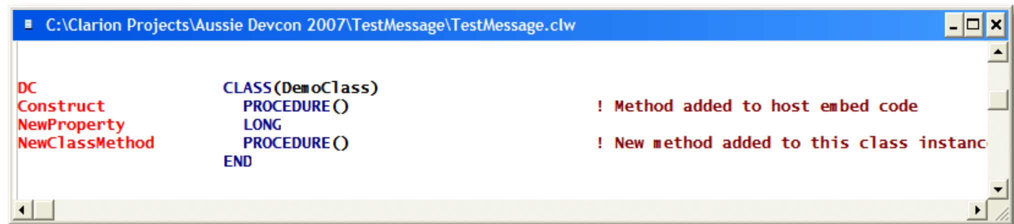

Figure 16 - The extended DemoClass

The new method and new property are clearly visible. What about the embed tree? Does it work there?

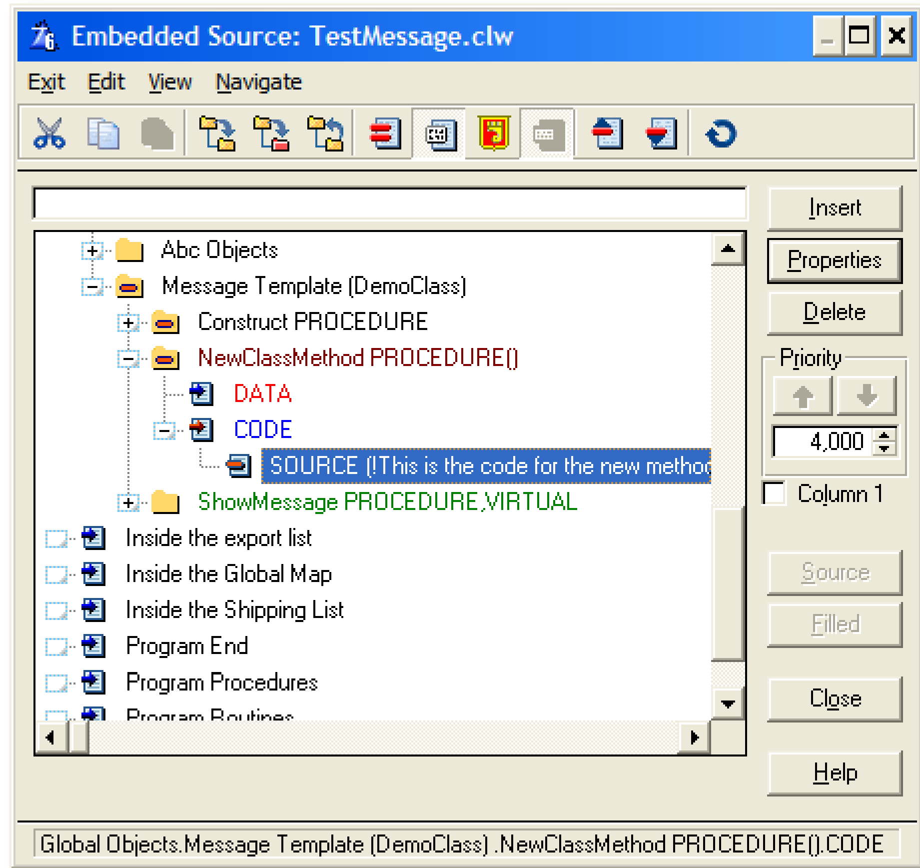

Figure 17 - Embed point for derived method

You can see the new method, in its own unique color (which is determined in the Clarion 6 IDE's setup area). The source is generated correctly too:

Figure 18 - The embedded source for the derived method.

Everything works and there was not much template code for you to write. This is fine for global template class wrappers, what about local wrappers? As you will soon see, this is nearly identical to the global template, but there are some differences.

Local Template Wrappers

There are times when a local class wrapper is required. Such cases would include #CONTROL templates like the ABC Browse control template. It is possible that you could have more than one browse list on a window. The underlying ABC classes keep the list populated with data; optionally use filters, range limits, various locator classes and other classes to make a responsive list control. This is where you see the default class name of *BRW* (for Browse) followed by a number.

All of the template code discussed so far is needed, yet there are some subtle changes required to make local classes work.

Local Class Dialog

I showed the class dialog, where you could derive new methods and properties, change the base class used and so forth. What you could not do was change the instance name. Local instances could (and sometimes should) change their labels, especially when you have multiple instances of the same class. While it will compile and work, it can be confusing to a developer as it is not obvious which browse control is *BRW2*.

Instead of using %GlobalClassPrompts, use %ClassPrompts instead. Both of these #GROUPs insert %CommonClassPrompts which checks if you are using a global dialog or not. If so, the %ObjectName entry is disabled.

Setting Default Instance Names

You do this in the #PREPARE and #ATSTART sections by calling %SetClassDefaults. In the global areas, I hard coded the class instance name. I could (and should) use symbols. Such symbols would be the result of an expression if you wish. However in the local areas, there is a new symbol that can distinguish the different instances. This symbol is named %ActiveTemplateInstance. To give you an idea of how this works, this is how ABC gives different instances of browse classes:

```
#CALL(%SetClassDefaults, 'Default',
'BRW'&%ActiveTemplateInstance, %BrowserType)
```

What about Embed Points?

This is something you should not overlook. This is a bit trickier than a global embed, mainly because you need to ensure that each instance has their own embed points and not one. If you add embed code in one instance, you may not be altering the behavior to the correct instance.

Again, the %ActiveTemplateInstance symbol comes into play. Here is how ABC handles the embed points for a given instance of a browse for specific methods (wrapped):

```
#AT(%BrowserMethodCodeSection,%ActiveTemplateInstance,'ApplyRange'
,'(),BYTE'),PRIORITY(4000),DESCRIPTION('Asign Higher Key Values
based on browse conditions')
```

For each browse control template you add to a window, there will be an embed point for each control. This even works when you change the name of the instance in the local class dialogs. The original name is never forgotten by the templates, thus nothing gets lost.

What Was Not Discussed

What I've laid out here is only things needed for ABC compliance. I've deliberately left out all the displays and prompts as they don't deal with ABC compliance; they help the developer use the template. This is up to the template designer what to add that is relevant in addition to the wrapper portions.

This talk has a narrow focus; what is only needed to make something ABC compliant? Templates, as a subject, are quite vast and full of little techniques that make template coding difficult. In order to keep those issues from intruding into this talk, they are deliberately left out.

To include these parts of template programming (and they are important), would hide how little code you really need to make an ABC compliant template wrapper.

Useful Things

I've included as part of this talk a skeleton template that you may use to make your own wrappers. You are more likely than not, need to insert meaningful strings and prompts on dialogs to guide developers on the best usage of your template wrapper.

Remember, a proper template wrapper of a class is an easy to use interface between the developer and your class.