Reborn Free

published by CoveComm Inc.

Search

Clarion MAGAZINE

August 2000 Index

code. Part 3.
(Aug 16,2000)

Code Documentation: The Achilles´ Heel Of Clarion
Documentation is essential for maintaining an application. Unfortunately there is no way to automatically document code in Clarion, and manual documentation is laborious and prone to errors. Stefan Rasmussen outlines a systematic approach to documenting your code.
(Aug 8,2000)

Displaying Related Fields In ABC EIP
Alan Telford explains how to display related fields when using ABC Edit-In-Place.
(Aug 8,2000)

Legacy to ABC: There is Another Way! Part 2
Daunted by the challenge of migrating your apps from Legacy to ABC? Simon Brewer shows how to do it one piece at a time with a hybrid of Legacy and ABC code. Part 2.
(Aug 8,2000)

Reborn Free CLARION online

published by CoveComm Inc.

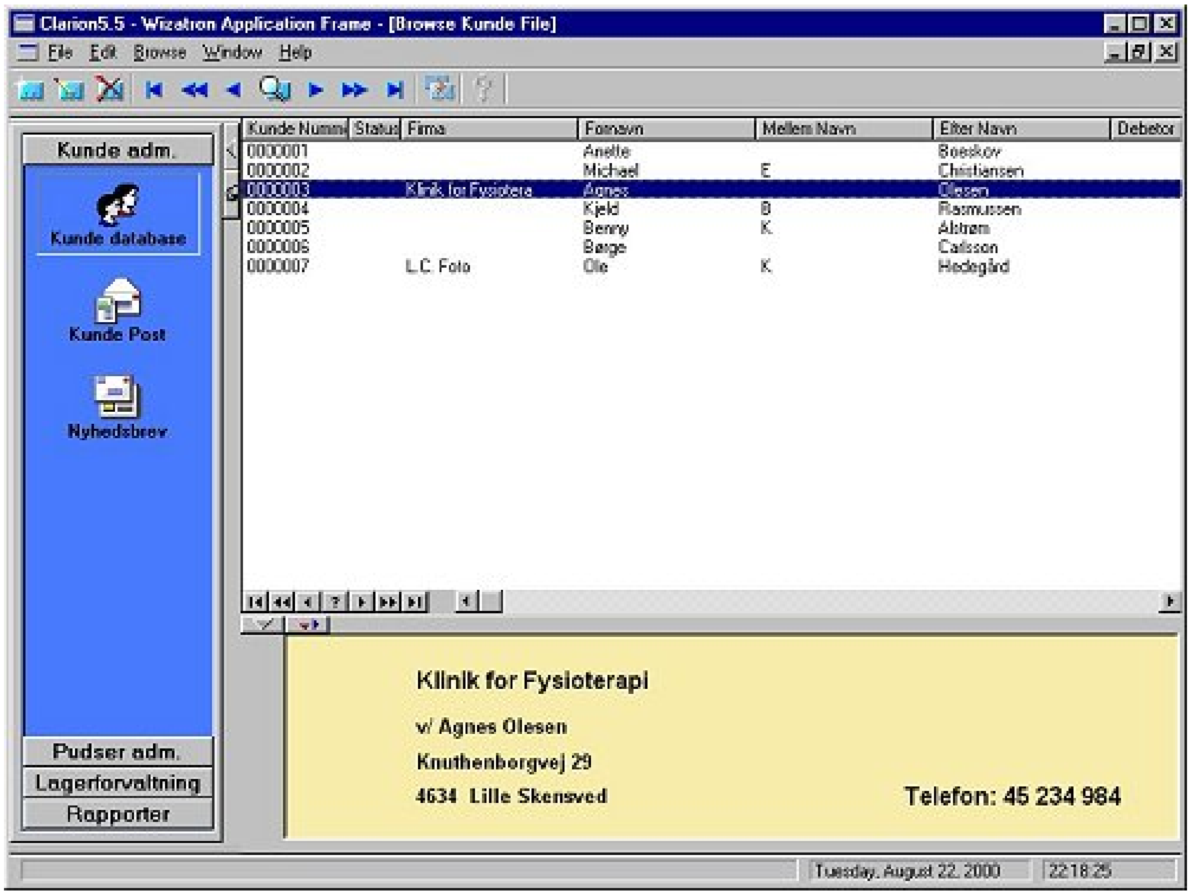
Clarion MAGAZINE

Creating An MS OutLook-Style Menu In Clarion

by Steffen Rasmussen

Microsoft Outlook uses an innovative menu style that's become quite popular. A while ago I decided to try creating this kind of menu for my Clarion apps. At first I thought it would be easiest to have the menu be a part of every procedure. As the application grew so did the workload, and making changes to the overall structure of the menu forced me to change every single procedure where the menu was implemented to keep the same look and feel. So it turned out that the "easy" way actually was very hard and time consuming. What I needed was an independent procedure which could contain this menu and control the calls to the other procedures. In this way I would only have one procedure to edit when I needed to make changes in the menu. A really great spinoff is that by making the menu in this way I can still use all the different templates and wizatrons that come with Clarion and third party products.

Figure 1. An application using an MS Outlook style menu.


Clarion5.5 - Wizatron Application Frame - [Browse Kunde File]
File Edit Browse Window Help
Kunde adm.
Kunde database
Kunde Post
Nyhedsbrev
Pudser adm.
Lagerforvaltning
Rapporter

Kunde Numm	Status	Firma	Fornavn	Mellem Navn	Efter Navn	Debetor
0000001			Anette		Boeskov	
0000002			Michael	E	Christiansen	
0000003		Klinik for Fysiotera	Agnes		Olesen	
0000004			Kjeld	B	Rasmussen	
0000005			Benny	K.	Alstrøm	
0000006			Børge		Carlsson	
0000007		L.C. Foto	Ole	K.	Hedegård	

Klinik for Fysioterapi
v/ Agnes Olesen
Knuthenborgvej 29
4634 Lille Skensved
Telefon: 45 234 984

Tuesday, August 22, 2000 22:18:25

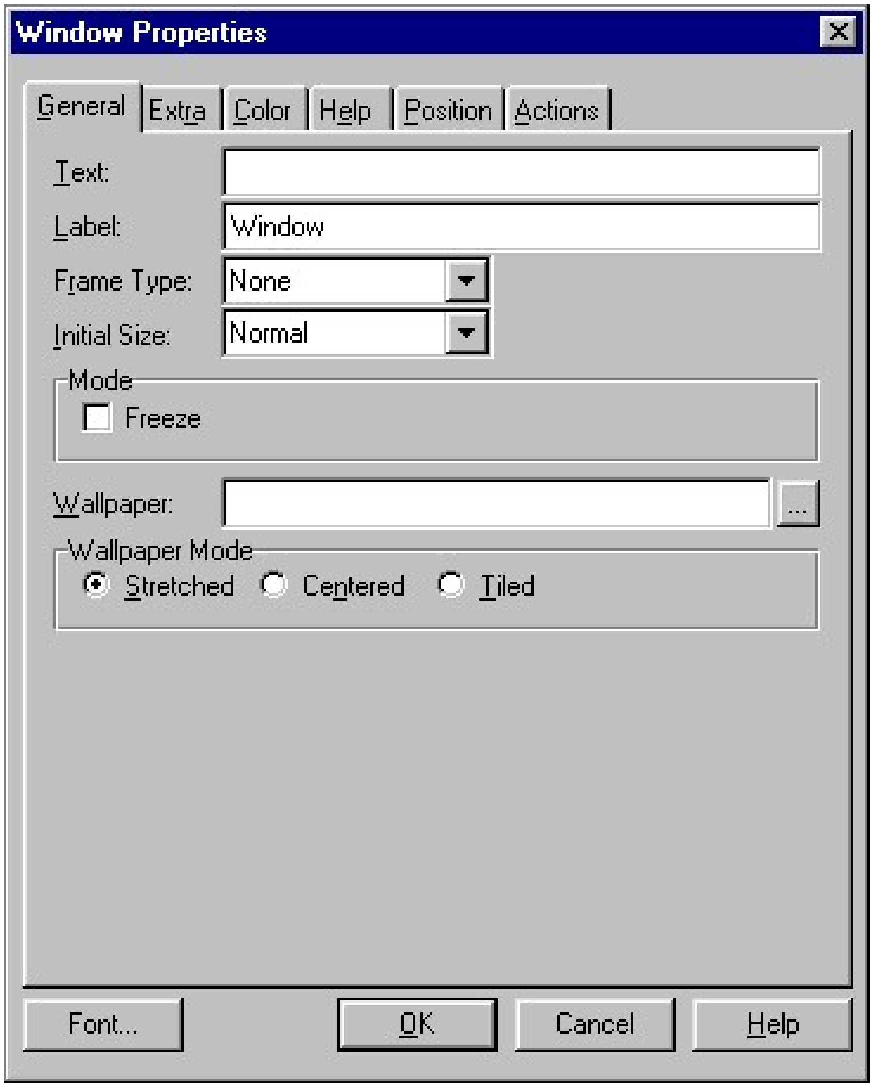
Designing The Menu

The basis of this menu is a Window procedure. In this example I will call it `MainMenu`. To create it use the Window – Generic Window Handler template. If you're following along you will now have an empty window with a title bar called "caption."

Right click the window and select properties. In the window properties you have to make some adjustments, since this menu has to be an integrated part of the application.

First of all remove the window title bar by deleting the text in the General field. The Frame Type is None and Initial Size is Normal.

Figure 2. Window General properties


Window Properties

General | Extra | Color | Help | Position | Actions

Text:

Label: Window

Frame Type: None

Initial Size: Normal

Mode

[ ] Freeze

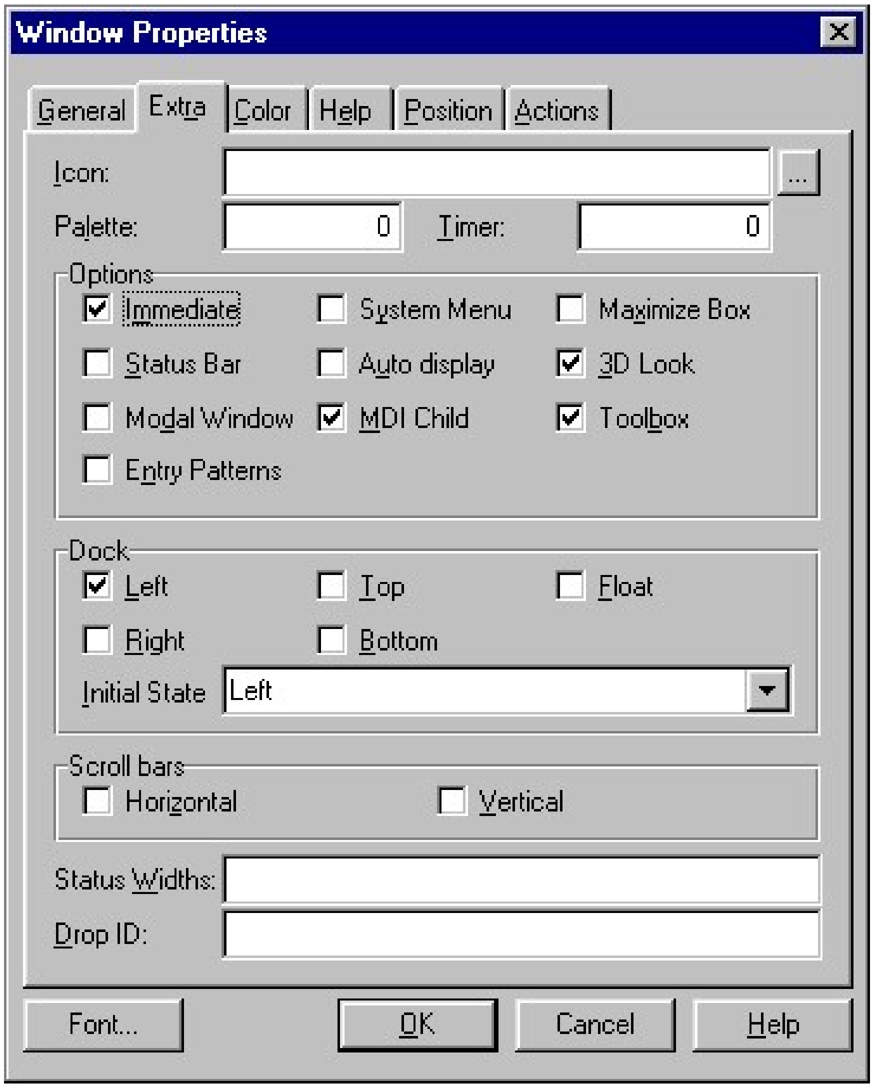
Wallpaper:

Wallpaper Mode

(●) Stretched ( ) Centered ( ) Tiled

Select the Extra tab and set the Options and Dock state as shown in Figure 3:

Figure 3. Window Extra properties.


Window Properties

General | Extra | Color | Help | Position | Actions

Icon:

Palette: 0 Timer: 0

Options
- [✓] Immediate
- [ ] System Menu
- [ ] Maximize Box
- [ ] Status Bar
- [ ] Auto display
- [✓] 3D Look
- [ ] Modal Window
- [✓] MDI Child
- [✓] Toolbox
- [ ] Entry Patterns

Dock
- [✓] Left
- [ ] Top
- [ ] Float
- [ ] Right
- [ ] Bottom

Initial State: Left

Scroll bars
- [ ] Horizontal
- [ ] Vertical

Status Widths:

Drop ID:

Font... | OK | Cancel | Help

Press Ok to accept the window properties changes.

The next step is to populate the window with the elements needed for the menu. Since this is an overall menu which is going to contain links to just about every procedure in the program, you will need a sheet where you can group the menu elements into different categories. You can call this `?SheetMain`, where each tab contains its own set of categories. Make the tab location below with joined scroll buttons.

You're also going to need two buttons. One for minimizing and maximising (`?ButtonResize`) and another for floating and docking (`?ButtonDock`) the menu.

When resizing the menu there will be quite a few elements that have to be hidden and unhidden. Instead of programming this functionality into every object you might as well place these objects into one group (`?GroupHide`) to hide and unhide them all at once.

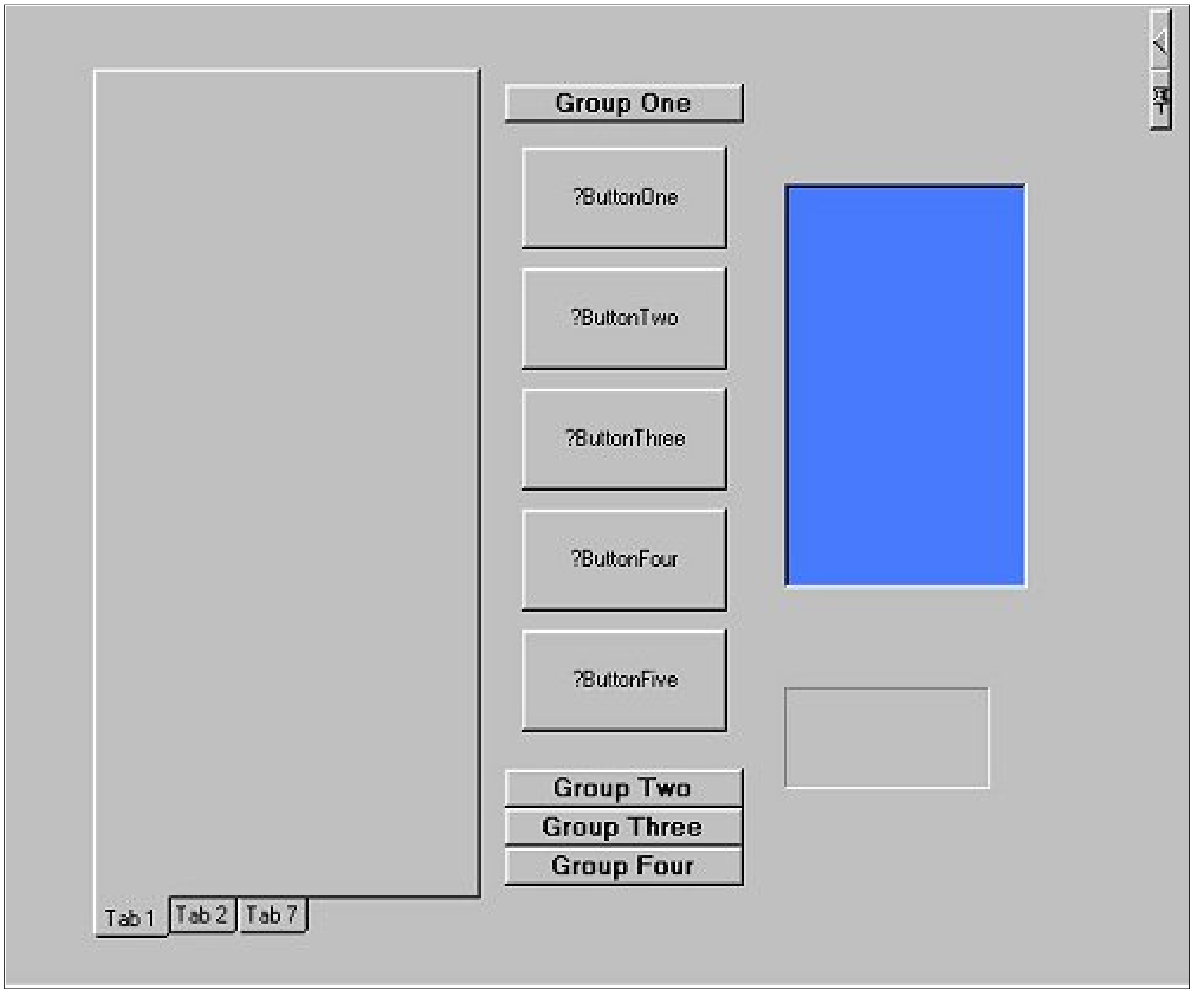
Now for the objects that will have to go into `?GroupHide`. Place within this group a panel (`?Panel1`) with an outer bevel of –1 and an inner bevel of –1. This panel is going to be used to colour the background of the different menus.

Apart from the panel there are also going to be five procedure selection buttons. Make these buttons flat. Each button will be given the name of the group and a sequential number from 1 to 5 and the default text name will be deleted. In a later article I will come back to these buttons to assign a button name, an icon and a calling procedure. Apart from the buttons there is also a control group box with the exact same size as the buttons, and with a Bevel outer –1, within each tab. This group is going to create the illusion that the button is depressed so the user can see which button has been selected.

The observant reader would probably suggest using flat radio buttons instead. This could have been used and it would save some programming but the problem with flat radio buttons is that when selected they "reduce" the colour of the button. On a grey background I think this is okay, but when using other colours it changes the overall look and the colour difference is too much. So by just using the boxed group I get the illusion I want with no colour changes.

That's nearly it for the objects. The last thing is placing four selection buttons. These selection buttons are going to represent four different menu groups which will contain the procedure buttons.

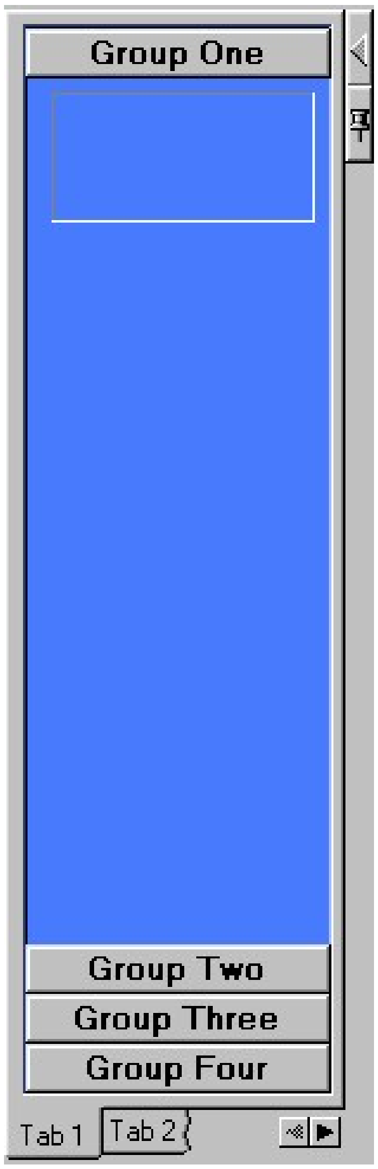
Figure 4. The window menu elements



Now that all the menu elements are placed in the window you just have to rearrange them so it begins to look like an OutLook menu. You have to keep the elements in their intended position within the group, sheet, tab or window. One way to do this is by assigning the object an exact position within the objects properties position tab. An alternative way is to drag and drop the objects. When doing this the Move Control in Clarion pops up and asks "Move Button into Tab?" Since you just spent a lot of time arranging the elements within the sheet and the group and presumably don't want to have to do it all over again, you answer No.

To make it easier to make future adjustments to the menu make the `?GroupHide` so small that visually its elements are outside the group. To prevent the elements moving from their position when repositioning the group you will have to resize the group. So if you want to move the group to the left resize it by making it wider and placing its left border where you intend it to be. Now just take the right border and resize it into position.

Picture 5: The rearranged Window menu elements



Now it looks much more like a menu.

Implementing The Menu Functionality

I could choose to have just the OutLook menu on the left side of window, but in some circumstances this has some disadvantages. Some users don’t like the OutLook menu so they have to have the option to turn it off. Others have a small screen resolution (for example 600 x 480 pixels) and therefore don’t have any room for this menu even if they like to, and so forth.

In an attempt to satisfy the different user needs I have implemented two extra buttons, `?ButtonResize` and `?ButtonDock`, which you won’t find in the OutLook menu. Clicking on `?ButtonResize` minimizes the menu and changes the button function to a maximizing button. To tell the difference between these two states the icon is also changed. Likewise with the `?ButtonDock`. It also has two window states, float and docked left. By making the menu float, users with a small resolution can still have the menu, and when the menu is not used they can minimize it.

Because the window can be in these four combined states (maximized, minimized, float and docked left) you will have to save the state the window is in, e.g. when minimized and floating it should still float when it is maximized. To do this you have to create two local data variables for the procedure `MainMenu`:

```
Float Byte
MinimizedMenuBar Byte
```

For implementing this dock and float functionality into the `?ButtonDock` you will have to place the following source in `Control Events.?ButtonDock. Accepted`:

```
!IF menu is floating
IF Float = 1
  !Change icon
  ?ButtonDock{PROP:Icon} = 'PinUp.ico'
  !The window can only Dock left
  WINDOW{PROP:dock} = Dock:Left
  !Make the window Dock:Left
  WINDOW{PROP:docked} = Dock:Left
  !Set local variable Dock:Left
  Float = 0
  !If the menu bar is minimized
  IF MinimizedMenuBar = 1
    !Set position top left corner
    SETPOSITION(?ButtonDock,0,20)
  ELSE
    SETPOSITION(?ButtonDock,96,21)
    !Set position top right corner
  END
!IF menu is NOT floating
ELSE
  !Change icon
  ?ButtonDock{PROP:Icon} = 'PinDown.ico'
  !Menu is minimized left
  IF MinimizedMenuBar = 1
    SETPOSITION(?ButtonDock,0,20)
    !The window can only Dock left
    WINDOW{PROP:dock} = Dock:Left
    !Make the window Dock:Left
    WINDOW{PROP:docked} = Dock:Left
    !The menu bar has full size
  ELSE
    !The window can only Float
    SETPOSITION(?ButtonDock,96,21)
    WINDOW{PROP:dock} = Dock:Float
    !Make the window float
    WINDOW{PROP:docked} = Dock:Float
  END
  !Set local variable to float
  Float = 1
END
```

The `Float` variable is used when the menu bar is minimized. In the minimized state it is always docked left, but it still keeps its docked state in the variable `Float` so that when it is maximized it can retain its original state. The resize button also has to have some code that can maximize and minimize the menu when selected. In the embed list for `?ButtonResize` place the following source in `Control Events.?ButtonResize. Accepted`:

```
    !IF menu is floating
IF Float = 1
  !Change icon
  ?ButtonDock{PROP:Icon} = 'PinUp.ico'
  !The window can only Dock left
  WINDOW{PROP:dock} = Dock:Left
  !Make the window Dock:Left
  WINDOW{PROP:docked} = Dock:Left
  !Set local variable Dock:Left
  Float = 0
  !If the menu bar is minimized
  IF MinimizedMenuBar = 1
    !Set position top left corner
    SETPOSITION(?ButtonDock,0,20)
  ELSE
    !Set position top right corner
    SETPOSITION(?ButtonDock,96,21)
  END
!IF menu is NOT floating
ELSE
  !Change icon
  ?ButtonDock{PROP:Icon} = 'PinDown.ico'
  !Menu is minimized left
  IF MinimizedMenuBar = 1
    SETPOSITION(?ButtonDock,0,20)
    !The window can only Dock left
    WINDOW{PROP:dock} = Dock:Left
    !Make the window Dock:Left
    WINDOW{PROP:docked} = Dock:Left
  !The menu bar has full size
  ELSE
    SETPOSITION(?ButtonDock,96,21)
    !The window can only Float
    WINDOW{PROP:dock} = Dock:Float
    !Make the window float
    WINDOW{PROP:docked} = Dock:Float
  END
  !Set local variable to float
  Float = 1
END
```

Previously I mentioned that the menu could have four different states, but as you can see from the code above it is actually only three. This is because when the menu is minimized it can only dock left and not float.

Reposition Menu Elements Upon Selection

Now for the menu selection functionality. When the user selects a group button all the group buttons have to move accordingly to make room for the selected groups procedure buttons. This rearrangement of the groups has to be triggered every time a group is selected and when the window is resized.

Instead of implementing this code in every category button you can place the code once in a routine (See ABC Embeds Are Easy for an explanation of routines) and then use it by calling it from each group button.

```
GLO:GroupNumber = 1 !Button Number selected
DO RoutineProcedureStart !Show active procedure
```

As you can see, this is just what is done in this example. The `GLO:GroupNumber (BYTE)` is assigned a value for the group button number and the second line calls a procedure `RoutineProcedureStart`.

```
Procedure Routines (MainMenu)
!******* Reposition the Group Buttons ********
!Last Number = Distance from bottom
?ButtonGroupTwo{PROP:Ypos} = 19
!Last Number = Distance from bottom
?ButtonGroupThree{PROP:Ypos} = 32
!Last Number = Distance from bottom
?ButtonGroupFour{PROP:Ypos} = 45

CASE GLO:GroupNumber
  OF 1
    ?Panel1{PROP:Fill} = 0FF7A48H    !Blue
    ?ButtonGroupTwo{PROP:Ypos} = (GLO:Height-58)
    ?ButtonGroupThree{PROP:Ypos} = (GLO:Height-45)
    ?ButtonGroupFour{PROP:Ypos} = (GLO:Height-32)
  OF 2
    ?Panel1{PROP:Fill} = 0C08000H    !Another Blue
    ?ButtonGroupThree{PROP:Ypos} = (GLO:Height-45)
    ?ButtonGroupFour{PROP:Ypos} = (GLO:Height-32)
  OF 3
    ?Panel1{PROP:Fill} = 0FF8080H    !Purple
    ?ButtonGroupFour{PROP:Ypos} = (GLO:Height-32)
  OF 4
    ?Panel1{PROP:Fill} = 0400080h    !Brown
  END
END
```

Here the group button is placed in its correct position depending on which group is selected (`GLO:GroupNumber`). Unfortunately this isn’t all. You have to take into account that the procedure buttons also have to relocate with the group buttons. So include the following in the same procedure:

```
!******* Reposition the Buttons acording to Group ********
SETPOSITION(?ButtonOne,,GLO:GroupNumber*13+10)
SETPOSITION(?ButtonTwo,,GLO:GroupNumber*13+50)
SETPOSITION(?ButtonThree,,GLO:GroupNumber*13+90)
SETPOSITION(?ButtonFour,,GLO:GroupNumber*13+130)
SETPOSITION(?ButtonFive,,GLO:GroupNumber*13+170)
```

What this code does is to take the group number which was assigned to the variable `GLO:GroupNumber` when the group was selected. This group number is than multiplied by thirteen, which is the height of the group selection button. If for example group three is selected these three groups will be stacked on top of each other. So now you have the total height of the groups. To this value you add the distance to each button.

Resizing A Toolbox Window

The Toolbox window unfortunately can't be resized with the normal Control Template. So you have to code the functionality into the procedure.

To start off you have to determine how high the program frame is at any given moment, as that determines the height of the toolbox. That means you have to transfer some data from the main frame procedure to the toolbox procedure. In the Global Properties select Embeds, and in After Global Data implement the following source:

```
EVENT:RefreshWindow EQUATE(EVENT:User)
```

Return to the Global Properties and select Data. Insert `GLO:ToolBoxThread`, `Long` and `GLO:Height`, `Short`.

Exit Global properties and in the embeds for Main(Frame) after `Local Objects. Resizer (WindowResizeClass). Resize PROCEDURE(). CODE`:

```
!Get height of Frame
GLO:Height=ThisWindow{PROP:Height}
!Height of Frame minus Menu and Toolbar
GLO:Height=GLO:Height-25
POST(EVENT:RefreshToolBox,,GLO:ToolBoxThread)
```

When a resize of the frame has occurred it posts the event refresh ToolBox to the `GLO:ToolBoxThread`. Now you have to return to the `MainMenu` procedure to catch this event.

First of all store the thread number of the toolbox window in `GLO:ToolboxThread` after the embed point Window `Events.OpenWindow`:

```
GLO:ToolboxThread =
Thread()
```

As well, `GLO:ToolboxThread` is reset when the window is closed. Window `Events.CloseWindow`:

```
GLO:ToolboxThread = 0
```

Next catch the event resize. After the embed point `Local Objects ThisWindow(Window Manager). TakeEvent PROCEDURE(). CODE`.

Parent Call:

```
IF EVENT() = EVENT:RefreshToolBox
   If Glo:ToolboxThread
      DO RoutineInitButton
   END
END
```

When the window is resized the height of the menu has to change accordingly, as doi the positions of the different buttons. Instead of using this embedded point for this task you might as well reuse the procedure you created previously and include the new window height code.

```
!**** Reposition the MainMenu ****
!Set size for window
ThisWindow{PROP:Height}=GLO:Height
!Set size for panel
?Panel1{PROP:Height}=(GLO:Height-23)
!Set size for sheet
?SheetMain{PROP:Height}=(GLO:Height-2)
```

As you can see in the source code the different OutLook style menu elements are given a height depending on the `GLO:Height` of the window Frame.

To make all this work it's very important that the frame font and size are the same as the `MainMenu` procedure. If they aren't the menu size and position will not be as intended.

Starting The Menu

In order to make the menu a part of the initial window (Frame) upon start up, you have to call `MainMenu`. This procedure call has to be initiated right after the program starts in the Main(Frame) embed point, `Window Events.OpenWindow`:

```
START(MainMenu,5000) !Start the procedure MainMenu
```

Now compile and take a look.

As you can see there is still some work to do. In the next article I'm going to look at how to dynamically change the menu procedure calls.

Download the source

Steffen S. Rasmussen has graduated in Computer Science from Copenhagen Business College. Since then he has worked as a programmer, system technician and network administrator, and is currently IT manager. Clarion is a quite a new language to Steffen since his only been working with it since January 2000. But what better way to learn it than by trying to teach others! Steffen has also set up a web site to collect as many examples of different user interfaces as possible to inspire Clarion developers.

Clarion MAGAZINE

Five Rules For Managing Complexity

by Tom Ruby

Part 2

When you are designing the database, or the data dictionary, you are actually constructing your program’s model of the universe. Since the entire universe is awfully big, you concentrate just on the tiny part of the universe that your program deals with. The more realistic your model of the universe is, the more able your program will be to do its job and deal with things you didn’t think about at the start.

In Part 1, I introduced the idea of sticking to rules in order to reduce the complexity of your application and to speed up development. I presented Rule Number 1 and two guidelines. To summarize Part 1:

> **Guideline 1:**
> Don’t take shortcuts. They might save you a few minutes now, but they’ll cost you days later.
>
> **Guideline 2:**
> It is a lot less painful to fix a painful mistake now than it will be later on.
>
> **Rule Number 1:**
> Eliminate repeating fields.

In Parts 2 through 5, I’ll continue to introduce rules (five in all besides the guidelines), show you why they are valuable, and explain how to apply them in your Clarion projects.

You can make up rules for all sorts of things. I’ve seen rules about how big buttons should be and where they should be located. I’ve seen rules about how lists should be displayed and about what fonts to use. Many of these serve to enforce a standard look and feel to the application and to save the developer time in deciding how to arrange a window. One rule I like to use is:

> **Guideline 3:**
> A list is resizable, a form is not.

Now, this isn't a hard and fast rule that says "never make a form procedure resizable," but it is a principle to guide me in deciding how or if to make a window resizable. You see, when a user stretches a list, they obviously want to see more entries. But if they stretch a form what do they want to see? Do they want to see more blank space? If the form is mostly a list, then it's obvious they want to see more entries in the list. The edit-in-place example from Part 1 contains a form that might be resized.

Anyway, enough procrastinating. Here's Rule Number 2:

> **Rule Number 2:**
> Eliminate redundant data

To explore this rule, I'll concoct a ridiculous example. Herb is writing a program for a garage. It is a really big garage with dozens of mechanics, and he would like to keep track of who is skilled at what sort of job. Herb might think of a table like this:

```
Mechanic Table
NameMechanic
DateBirth
SocialSecurity
RateHourly
Skill
```

But, of course, a mechanic probably has more than one skill. A common solution is to just list the mechanic multiple times in the table. But now Sue Pipebender is getting married. She has 241 entries in this table, and somebody has to go through all those entries and change her name. Oh well, what are secretaries for anyhow?

Herb might make his table like this:

```
Mechanic Table
NameMechanic DateBirth SocialSecurity RateHourly Skill1
Level1 Skill2 Level2 Skill3 Level3 Skill4 Level4
```

But Rule Number 1 tells us to avoid this sort of thing, so Herb builds two tables:

MechanicTable	**SkillTable**
NameMechanic DateBirth SocialSecurity RateHourly	NameMechanic Skill Level

So now each mechanic has a list of skills. This looks pretty good, except he wants to be able to search by skill, and somebody will likely enter, "Wheel Alignment" for Crusty McGinnes, "Alignment" for Joe Schmoe and "Align" for Guido. They could write the "approved" skill names on a piece of paper and tape it to the monitor but hey, this is the 21st century (well, almost). Use a Post-It instead!

Herb, you can do better than that. Make a table of skills so you can put a drop list to pick the skill from. So now the tables look like this:

MechanicTable	**MechanicSkillTable**	**SkillTable**
NameMechanic	NameMechanic	Skill
SocialSecurity	Skill	
DateBirth	Level	
RateHourly		

There's nothing really wrong with this setup except when Herb changes a mechanic's name, he has to change the names of all the related records in the middle table. Using referential integrity, Clarion will take care of this for you, except…

Clarion always edits a record in memory and writes the record to the table when the user completes the form. If you put the list of skills on the mechanic form, when you go to change Sue Pipebender's name to Sue Arcwelder, her skills disappear! That's because there aren't any records that say Sue Arcwelder in the `MechanicSkill` table. They come back when you save the record and the record is actually changed causing the RI code to change all the Sue Pipebender entries in the middle table to Sue Arcwelder. The users won't like this at all. What is Herb to do?

Look closely at Herb's `MechanicSkillTable`. It lists the mechanic's name and the name of the skill, but it doesn't really have to. The middle table's job is just to say "this mechanic has that skill," and it doesn't really care what the mechanic's name is. So here is where Rule Number 2 comes in. The purpose of Rule Number 2 is not really to reduce the storage required, but to make updates happen efficiently without surprises. I'll rearrange Herb's tables just a little.

MechanicTable	**MechanicSkillTable**	**SkillTable**
SysIDMechanic	SysIDMechanic	SysIDSkill
NameMechanic	SysIDSkill	NameSkill
SocialSecurity	Level	
DateBirth		
RateHourly		

I added a hidden field to the mechanic table and to the skill table which is used solely to link (or relate) them to the MechanicSkillTable. Notice I said it's a hidden field. I never display this field! Why? The user doesn't care what the SysIDMechanic is, so why show it to her? Also, these hidden fields don't mean anything so they never have to be changed. It might be tempting to use the mechanic's social security number for this linking field, but social security numbers sometimes change, particularly when they were mistyped in the first place. The Social Security Number might be an important key field for looking up a mechanic's record, but don't use it as your primary key.

Sometimes, people suggest making the Social Security Number unique will fix the problem, but what if you're entering a new employee record and a previously entered record has an incorrectly entered number and it's causing the new record to be rejected? Hey, I saw it happen, but it still didn't convince the DP department to use a hidden key field.

Sometimes this hidden field is called a "surrogate key."

A field which is meaningless to the user never has to change. I'll express this in Guideline 4:

> **Guideline 4:**
> Link your tables by a hidden field that is completely meaningless outside the system.

So what does this gain? For one thing, updating the mechanic's name is now instantaneous. No other records have to be updated. Sure, Herb could buy a power server to hold these tables and run a fast data engine on it, but can even a power server update a bunch of records faster than it can update no records?

I went ahead and used a hidden linking field for the skill. Why? For the same reason. Supposing somebody mistyped "Water Pump." To correct it, they just have to look up "Water Pmup" in the skill browse and fix it. Immediately, everyone who was skilled in "Water Pmup" becomes skilled in "Water Pump." By the way, a form on this skill browse listing all the mechanics who have a skill would be a really convenient place for the users to ask, "Who can balance wheels?" When the garage wants to go to skill codes, all they have to do is edit the skill names, and instantly, all the mechanics that know how to "Flush Cooling System," have skill code 2432. I don't know why the garage would want to do this. Perhaps a skill code sounds more computerish.

"But I want to show the name of the skill on the browse, not some stupid `SysID`." Indeed! I don't want to show the `SysID` on the browse either, in fact, I never want to show the `SysID`. When you build your tables in the dictionary editor or whatever your favorite tool is, specify a key for each of your linking fields. Then specify a relation between the tables.

Always put your relation in the dictionary. "But there's business reasons not to relate these two tables." Nonsense. The tables and relationships are describing the world to the program. These business reasons sited are actually business rules that limit how do you things and they are reflected in the program code, or in the procedures and triggers of your SQL database.

Here is the rule I use to make keys:

> **Guideline 5:**
> Use keys to help the application identify records it is interested in.

There are two places where Guideline 5 applies. First, you make your linking fields, which I usually call `SysIDs,` into keys. This lets the application quickly find all the `MechanicSkill` records that apply to a mechanic, and the `Skill` record that applies to a `MechanicSkill` record. Second, you make a key (or an index if you're dealing with SQL) to help a browse identify "the next eight records to show."

Now that you have a relationship between the tables, it is trivially easy to make the browse show the skill name instead of the skill `SysID`. You just add the `Skill` table under the `MechanicSkill` table and pick the field that contains the skill name. The relationship tells the templates how to connect the `Skill` with the `MechanicSkill`. The same thing applies to reports. If you want to make a report showing all the skills each mechanic has, just put the three tables in the file schematic. Somehow, looping through the child records on a report just became pretty easy.

Sometimes, data is not redundant but looks like it is. Think about a sales and inventory system. There might be three tables that look like this:

Customer	**Sold**	**Inventory**
SysIDCustomer NameCustomer	SysIDCustomer SysIDInventory DateSold Price	SysIDInventory Description Price

Obviously, the `Sold` table tracks who bought what and when, but what is that price field doing there? Wouldn't it be better to skip the price field in the `Sold` table and look instead at the price field in the `Inventory` table? No, actually they are two different price fields sloppily named the same thing. The tables should look like this:

Customer	**Sold**	**Inventory**
SysIDCustomer NameCustomer	SysIDCustomer SysIDInventory DateSold Quantity PriceSold	SysIDInventory Description Price

You see, the `Inventory` record shows what the store is asking for the item now, but the `Sold` record shows what the store was asking for the item when it was sold. Supposing the storekeeper wanted to keep more information about the price history? You might build him four tables like this:

Customer	**Sold**	**ItemPrice**	**Inventory**
SysIDCustomer	SysIDCustomer	SysIDItemPrice	SysIDItem
NameCustomer	SysIDItemPrice	SysIDItem	Description
	DateSold	DateStarting	
	Quantity	DateEnding	
		Price	

Now `ItemPrice` contains a history of the price of each item, with starting and ending date. I didn't put the Price Sold in the `Sold` table because that can be found in the ItemPrice record. This would work unless the store negotiates prices with the customer, in which case PriceSold would have to be added back to the Sold record.

To further illustrate, think about a company that ships things to its customers. Each customer has an address, so you might think you could skip putting the address in the shipment record like this:

Customer	**Shipment**	**ShipmentDetail**
SysIDCustomer NameCustomer CompanyCustomer Address City State Zip	SysIDShipment SysIDCustomer DateShipped	SysIDShipment SysIDInventory Quantity
Inventory SysIDInventory Description		

Be careful. This can only record shipments shipped to the customer's address. Perhaps the customer wants the shipment shipped to his client or his mother. Maybe the client moved. You probably want the data to look more like this:

Customer SysIDCustomer NameCustomer CompanyCustomer Address City State Zip	**Shipment** SysIDShipment SysIDCustomer DateShipped AddressShipped CityShipped StateShipped ZipShipped	**ShipmentDetail** SysIDShipment SysIDInventory Quantity
Inventory SysIDInventory Description		

Usually, the `AddressShipped` field is just copied from the `Address` field, but the customer might want it shipped somewhere else. Do you want your users telling their customers they can't do that just because the program can't accommodate it? This might seem like a lot of redundant data, but actually it isn't. Consider a customer that moves: you change the address in the customer record, but as you deal with them, you have a record that the TV they bought last year was shipped to 11535 IL HWY 9, while the monitor they just bought was shipped to 19215 N 100th Rd. This can be valuable information if a question comes up, and it's really cheap to store this extra address. Remember, remove redundant data to make updating it fast and reliable. Keep historic data when necessary.

You'll want to automate the process so the computer is printing the shipping label from the database. If a question comes up as to where the shipment went, you have the address it was shipped to right here.

Just be sure to automatically get all the information from the database to the shipping label! I'll illustrate this with a true story. Most of the names have been omitted to protect… you know the drill.

A client wanted to ship me a piece of equipment to test a new program on, so he went to the manufacturer's online ordering web site and entered the order. The order never came. The manufacture's customer service department had to go searching all over. My client had entered the order as:

Name:	Tom Ruby
Company:	His Company Name
Address:	19215 N 100th Rd
City:	Industry
State:	IL

But the clerk looking at the screen, typed the order into their fulfillment system like this:

Company:	His Company Name
Address	129215 N 100th Rd
City	Industry
State	IL

Now, Industry is a small place, and the UPS driver knows me pretty well. If the equipment had been shipped to Tom Ruby at the incorrectly entered address, the UPS driver would have figured it out. Or, if it had been shipped to His Company Name at the right address, the UPS driver would have figured it out. But with neither my name, nor my address, the equipment wound up at the receiving department of a coal mine for several days while they scratched their heads wondering who would have ordered something from His Company Name. The shipping department had confirmation that the item had been delivered to 129215 N 100th rd and thought all was fine. The order entry system had recorded that the item was shipped to 19215 N 100th Rd, and all was fine. Since everything was fine, the manufacturer was at a loss to figure out why I didn't have it. Had the shipping address been automatically copied to the shipping label, there wouldn't have been a problem.

Another example: you would think that the big credit-card issuing banks with their huge DP departments would get their data design right, but alas. Have you ever lost a credit card? They have to close the account and open another one just like it, and you'll get two bills, one showing charges and payments made on the old account, and the other showing charges and payments made on the new account. Do you see what the problem is? (Yes, you in the back.) That's right: the card number is the primary key and all the transaction records use the card number to connect back to the account record. If they used a surrogate key to relate everything back to the account record, all they would have to do is set the card number field of the account record to the new card number. Probably they want to keep the card numbers in a separate table so they can record that this card belonged to that account but was stolen.

So to recap, here are the two Guidelines and Rule Number 2:

> **Guideline 3:**
> A list is resizable, a form is not.
>
> **Guideline 4:**
> Link your tables by a hidden field that is completely meaningless outside the system.
>
> **Guideline 5:**
> Use keys to help the application identify records it is interested in.
>
> **Rule Number 2:**
> Eliminate Redundant Data

Next time, Rule Number 3.

Tom Ruby, who is no relation to the man who shot Lee Harvey Oswald, is an independent contractor living in the middle of a hayfield in Central Illinois with his wife Susan and two red-headed sons, Caleb and Ethan. He has been using Clarion for Windows since the summer of '95. Before that, he was a "TopSpeeder" using Modula II, so he has never used the DOS versions of Clarion.

Reborn Free CLARION online

published by CoveComm Inc.

Search

Clarion MAGAZINE

Clarion News

August 29, 2000

Creating An MS OutLook-Style Menu In Clarion
(Aug 29,2000)

Five Rules for Managing Complexity: Part 2
(Aug 29,2000)

August 2000 News
(Aug 30,2000)

BitList Template Update
(Aug 29,2000)

Read The July 2000 News

Buggy 1.1 Available
An update to Buggy, the Bug tracking tool, is available to all registered users. The major improvement is the ability to automatically send EMails to customers and responsible persons upon user-definable events.

SetupBuilder 3.10 Coming Soon, Price Increasing Sept 1
SetupBuilder's price is increasing from $119 to $149 on September 1. Purchase SetupBuilder 3.0 now for $119 and you will get a free update to the new version (of course, all registered SetupBuilder 3.0 users will get a free update). Major enhancements in 3.10 include support for unattended installations, Spanish, Russian, and Portuguese language modules, Windows and Clarion version condition files, support for administrator privileges on Windows NT and 2000, NT4 Service Pack 6a and Windows 2000 Service Pack 1 detection, multiple OS version checking, removal of "in-use" files, and more Clarion code examples.

Gitano Software C5.5 CR1 Compatibility List
The following Gitano utilities are compatible with C5.5 CR1 and are now available for download: G-Cal, G-Calc, G-Notes, G-RegPlus, G-RegDistributor, and G-Buddy

UltraTree Platinum Premium 5.7 Adds Style Sheets
UltraTree Platinum now fully supports listbox text style sheets. You can now use a different font in each section of the tree, use heading styles for headers, normal styles for the data, or even mix fonts in the same row. Styles can be customized by end users. No coding required. Style sheets require Clarion 5 or Clarion 5.5, and are exclusively a premium feature. Platinum Premium also supports tagging of records and tagging of subtrees, and will export tagged rows for reporting. Clarion 5.5 Candidate Release 1 is supported.

CPCS Faxing v1.0 Released
CPCS has released the CPCS Faxing AddOn v1.0.The Faxing AddOn allows you to fax CPCS reports directly from your program to one or more recipients. Faxing is performed by WinFax PRO v9.03 or later (a copy of WinFax PRO is required on each machine that needs to fax). A demo program, and help file can be downloaded from the CPCS website. The Faxing AddOn is $129.

Wild Wild Wares Templates Updated
New templates, updates and demos at Wild Wild Wares include enhancements to the Hover control, registry PutReg and GetReg functions, windows message subclassing, screen saver enable/disable, style sheets, debug templates, bug reporter, and more.

Clarion Third Party Profile Exchange Updated
The Clarion Third Party Profile Exchange now contains 202 product profiles, and 165 vendor profiles. New in this edition, the CPCS Faxing AddOn and the Winword Previewer.

TopSpeed

Clarion 5 by TopSpeed

FREE Microsoft Internet Explorer

etc2000 EVENT SPONSOR

DeveloperPlus
Your source for development tools and add-ons.
Your outlet for application sales.

August 22, 2000

James Fortune's ETC Presentation Now Available

James Fortune has set up a page for his ETC2000 presentation on writing (or not writing) help for applications. James' PowerPoint presentation is available in two versions, with and without sound. Also on the site, more ETC pics.

Ragazzi Templates/Utilities Update

New versions of the Ragazzi templates, utility DLL, and Developer's Toolkit have been uploaded. These care compatible with Clarion 5.5, Candidate Release 1.

Queue Edit in Place Template Update

Keystone Computer Resources has announced an update to their Queue Edit in Place Template templates. Version 1.00.002 includes enhanced support for multiple Queue Edit in Place Controls and colored columns, and is compatible with Clarion5.5EE.

Nettools Enterprise Edition Update

Keystone's NetTools Enterprise Edition templates have been updated. Version 1.00.020 includes additional function prototypes, enhanced installation, and automatic template registration. Compatible with Clarion5.5EE.

ARCO Word Reporter Version 1 Released As Freeware

To celebrate the success of the current release of ARCO Word Reporter, ARCO Software has released version 1 as freeware. ARCO Word Reporter integrates MS Word as a reporting engine, with Word's full formatting power. Please note that ARCO provides no support for freeware versions of its software.

Gitano Software Survey

Gitano Software has a short online survey for Clarion developers. Results will be published after the survey is complete.

New Winword Previewer

Oleg Fomin, author of Fomin Report Builder, has released Winword Previewer, a template which allows Clarion applications to send reports to MS Word for previewing and minimal editing.

Goodhew Heads New Clarion Products/Services Company

Randy Goodhew, a long time Clarion software developer, author, and lecturer, along with a group of partners, investors and advisors, has formed eQuarion Corporation. "Our goals are to provide Clarion users with an array of products and services that will enhance their use of Clarion and to assure a growth path toward industry standards.", says Randy Goodhew. "It is vital that Clarion developers be able to co-exist in the current Windows environment of multi-language and component driven development." The company sees Clarion as a "primary or subordinate tool for developing mixed-language and database oriented applications," and is working toward partnerships with industry-standard development tool suppliers. Permanent offices will be located near Cincinnati, Ohio and the domain www.equarion.com has been registered for use as a website (now under construction for later this year). A future press release will be issued when additional information is available.

August 15, 2000

C5.5 Candidate Release Available
Clarion 5.5 Release Candidate 1 is now available for download. The beta still has some rough edges, but contains a number of new features like RTF and HTML help support. Frequent electronic releases are planned until all the bug fixes and new features have been delivered. Feedback is appreciated. To get the candidate release you fill out a user profile and the download instructions are mailed to you. This registration process was created using the as yet unreleased ASP templates! These templates are designed for high volume requirements and are getting some early testing as a large number of users are expected to download the candidate release. PDF manuals will be posted for download on Thursday. Clarion 5.5 Professional users will receive upgrade instructions within a few days.

New Clarion Web Ring
Gitano Software has created "The Master Web Ring", a conglomerate of resources for the Clarion community. Anybody that has anything of value to the Clarion community can be included. Content and link style guidelines apply.

LSZip Installation Changes
A version of LSZip is now available that does not require a password to be installed. Only the UserName and UserPIN are required. All known warez site keys are locked now.

UltraTree Platinum Update
UltraTree release 5.62 is now available for download by registered users. Tagging can now be enabled or disabled at individual row granularity. A virtual method is called during loading of each row in a tagging-enabled tree section. If it returns FALSE, tagging is disabled for that row. Several additional methods and properties support the feature. New methods return the "leafness" of the row whose number is passed as an argument. If runtime translation is in use, PopupClass.SetTranslator calls are now an optional feature. As well, the User Guide has been revised.

New Telephony Page
Craig Ransom has created a new "Lair Page" for telephony using ExceleTel's TeleTools. The page contains essential instructions for using TeleTools with Clarion 5b.

Silicon Raid Almost Free Bug Tracker Update
The link to download the "almost free" bug and defect tracker known as Silicon Raid has been updated. Shawn Mason is also looking for input on function point per bug/defect tracking.

WSpell Spellchecker Demo
Leonid Chudakov has made available a demo showing WinterTree's WSpell spellchecker ActiveX control. Requires 30 day trial version of WSpell.

August 8, 2000

Gitano Utilities Update
The Gitano G-RegDev, G-RegPlus, and GRDistributor utilities have been updated and are available for download. These are free updates to all registered users.

BaseCw International Version Updated
The new international version of BaseCw (V3.0 freeware) from Eric Griset is available. This product

manages mail and news message archives. If you were previously using BCW V2.0, you can also download a utility to convert your BCW2.0 database to V3.0.

SetupBuilder Language Modules Updated
The SetupBuilder 3.0 Installation System now supports the following languages: Danish, Dutch, English (US), French, German, Norsk (Bokmal), Russian, Spanish and Slovenian.

Insight 1.0 Beta 2b Released
Beta 2b of the Insight graphing product is now available. New in this beta is the ability to interact with the graphs; the user can change the graph type, print the graph, save it as a file, and so on. Drill-down graphs are also supported, as is auto-shading. This product is a Clarion DLL and not an OCX. Insight will usually cost $299, but will be priced at $199 for the duration of the beta program. Beta users get free upgrades to the gold release, and beyond.

NetTalk 1.0 Beta 8 Released
NetTalk Beta 8 is now available. There has been a lot of work on the SimpleConnect feature, and email support is now included, including MIME, SMTP and POP3. Documentation has also been overhauled. The normal price for NetTalk is $299, but it's currently on at $199 during the beta program. Beta users will automatically get a free upgrade to the gold release, and beyond.

Special Agent Version 1.24 Released
CapeSoft’s Special Agent has had a minor upgrade. This release focuses on reducing the amount of generated code, and better support for IMM windows. Compatibility with a number of third party characters is under development. Special Agent costs $199.

WinEvent Version 2.7 Released
WinEvent is a library that simplifies the creation of background applications. It also includes a number of functions for reading and writing to serial ports. This update adds some advanced Comms functions, which allow you to set the state of the hardware lines. In addition support for mouse clicks on the Icons in the System Tray has been improved. WinEvent costs $30.

CapeSoft To Support Clarion 5.5
CapeSoft will be recompiling all version-dependent DLLs as soon as Clarion 5.5 becomes available. Updated will be posted on the web site.

Bruce Johnson To Speak At Brazilian DevCon
CapeSoft’s Bruce Johnson will be attending the Brazillian Devcon August 21 and August 22, 2000, in Sao Paulo. For more information on the Devcon contact Sergio Baratojo at sergio.baratojo@lifetech.com.br.

solid.software Office Closed For Holidays
The solid.software office will be closed from August 7 until August 20. All orders and support mails will be processed as soon as Chris & co. are back from holidays in Switzerland.

DeveloperPLUS Introductory Pricing To End Soon
Lee White’s DeveloperPlus, an e-commerce and software delivery service for Clarion Developers, will end its introductory pricing soon. DeveloperPLUS provides enabling technologies and services that allow developers to offer their products online without the costs and overhead involved with sales and fulfillment. Customers can pay by Visa, MasterCard, Amex, Discover and Novus.

RemFlash Goes Gold
RemFlash a reminder template which also provides "Instant Messenger" capabilities when used in a networked configuration. Features include easy implementation of reminders, visible warnings when a reminder is due, option to beep or play a WAV, and complete or reactivate a reminder. Priced at $149 with all source and no runtime royalties.

Silicon Raid: Free Bug And Defect Tracker
Silicon Raid is a mostly free bug and defect tracker. Features include: per program/per revision tracking, unlimited queries/reports, revision report for manuals, help files etc, replication of bugs, PDF/DOC manual. Free for one person shops, extra licenses are $59.95. The install password is "YO". The initial login is "1" and the password is "1" (for those who won't read the install screens).

Clarion Prize Draw

Sterling Data is holding a free prize draw worth up to $195 through August 13, 2000. The prize is any one of LogFlash, SearchFlash, IMPEX, CopyFlash, BackFlash, and RemFlash.

INVEST in your own abilities Clarion magazine

published by CoveComm Inc.

"Error! Error! That Does Not Compute!"

by George Cobaugh

> **NOTE:** Clarion Magazine was not able to obtain an archive of the original source code for this article. If you have the source, we'd very much appreciate it if you would email it to editor@clarionmag.com so we can post it for other readers.

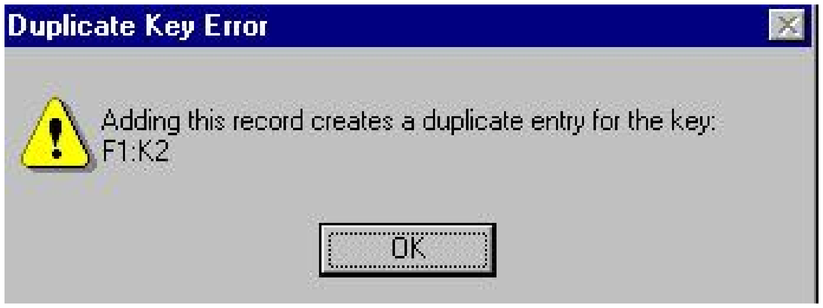
We've come a long way from the sixties, where this famous quote from a popular television series was the type of error message we expected from a computer (at least when we were kids). Now, we are application developers and we have to think of the end users when we process or trap errors. Also, when we get a support call and this is the error message that is relayed to us from our program, we don't have a clue what might be the problem. Then we sound pretty lame when we inform our customer that we don't know what this error means.

In the past, we have mostly relied on Clarion to process our errors for us. A lot of times I have been asked the question: How can we change the error messages that are generated by the templates and runtime library in Clarion? My answer would always be: Change the templates (for part A) and you really can't (for part B). The end result for most of us would be to just 'go with the flow' and use the generated errors.

When we would trap our own application errors, it would be by providing a form of the **MESSAGE** statement. Sometimes **CASE MESSAGE** would be used to give the end user a choice of how to handle the error. Most of the time, we would put the same MESSAGE structure in several places in our programs with very little change in the text. Just type it over and over.

Now, we have an advantage of OOP and C4, we have an object called **GLOBALERRORS** that we can use to our advantage. Here is where Clarion processes the internal messages it generates, by making a call to this object. It is one of the Base Classes included with the ABC classes and is included in every program we write using the ABC templates. We can use this object like all the other objects in C4 ABC. We can change the properties and call the methods from anywhere in our application because this object is instantiated globally.

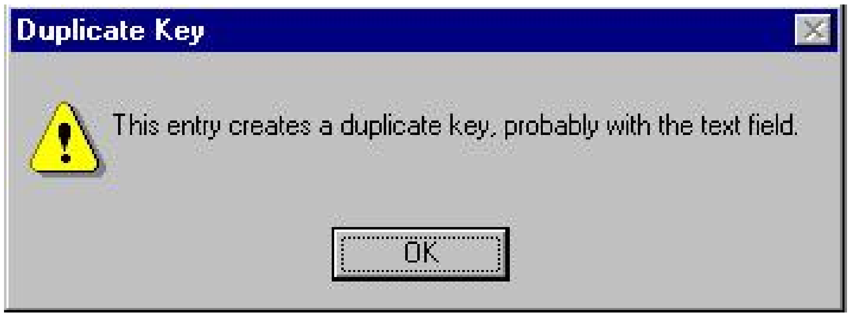
A common error that is processed by Clarion is the duplicate key error. If we use some cryptic naming conventions for the files, fields, and keys in our dictionary, the message generated can be a little confusing for our end users. For example, if we have a file named FILE1 with a key called F1:K2, this doesn't mean anything to anyone but us. The message generated by default would look like this:



Duplicate Key Error

Adding this record creates a duplicate entry for the key:
F1:K2

We would know that we have a duplicate key, and we, as the developer, would know which key it was, and which field probably caused it. Our end user, on the other hand, would probably not be able to know what they did wrong, or what data caused the error. At least until they have encountered this a time or two. On the other hand, if they saw this:



Duplicate Key

This entry creates a duplicate key, probably with the text field.

They would have a better chance of recognizing what it was that caused it, and be able to correct the situation on their own without calling us in the middle of the night and waking us up, and depriving us of our beauty sleep. It tells them that it was a duplicate field, and tells them which field is probably the duplicate. They only have to change the text field, and the problem goes away. This would require a template change in the standard templates. OOP makes this easier now.

Another function of the Error Class is the messages that are used for things like: The delete message, when we delete a record from the browse. The message to add another record if we have this selected in the Messages and Titles properties of our form procedures. The ToDo message for procedures we have not yet defined in our application. Etc... These messages would require a template change to change the messages in the CW2.003 (and before) standard templates. These would be permanent changes and would reflect in all of our applications from that point on, or we would have to use and maintain separate template sets for separate programs. Now, we have this functionality as an object. Since it is an object, we can change the properties at runtime, and call the methods when we need to use this object. OOP makes this easier.

Finally, we have our own application specific errors that were handled with the MESSAGE or CASE MESSAGE functions before. Guess what? That's right, OOP makes this easier. We can handle all of these things with the object GOBALERRORS that the ABC template so kindly creates for us globally.

I'm going to show you how we can use the 'out of the box' errors class to create our own custom error messages and change the existing default error messages by using this Class. It is really quite easy to do. Let's look at some of the things that have been provided for our enjoyment.

1. A data structure to contain the error ID, the severity level of the error, and the text for the caption bar and the message.
2. A list of common window and file errors that we can use and/or override with our own messages.
3. A set of macro symbols to facilitate the inclusion of information into our messages, such as %field and %file to allow for common messages with substitution of field names, and file names.

4. The methods to call to add, change, remove, and process the errors.

The Data Structure

```
AppErrors   GROUP
Number        USHORT(7)
              USHORT(Msg:RebuildKey)
              BYTE(Level:Notify)
              PSTRING('Invalid Key')
              PSTRING('%File key is invalid.')
              USHORT(Msg:FieldOutOfRange)
              BYTE(Level:Notify)
              PSTRING('Range Error')
              PSTRING('%Field must be between %Message')
              USHORT(Msg:MyError1)
              BYTE(Level:Notify)
              PSTRING('This is my error number 1')
              PSTRING('%Field is %Message')
              USHORT(Msg:MyError2)
              BYTE(Level:Notify)
              PSTRING('This is my error number 2')
              PSTRING('%File is locked. Wait a few minutes and '|
                 &'try again.')
              USHORT(Msg:MyError3)
              BYTE(Level:Fatal)
              PSTRING('This is my error number 3')
              PSTRING('This is a fatal error and will end the '|
                &'program')
              USHORT(Msg:UnknownError)
              BYTE(Level:Fatal)
              PSTRING('Unknown Error')
              PSTRING('This is an unknown fatal error '|
                &'and will end program')
              USHORT(Msg:ProcedureToDo)
              BYTE(Level:Notify)
              PSTRING('Process Not Completed')
              PSTRING('The process you have selected '|
                &'has not been completed yet. '|
                &'It will be available in a later update.')
            END
```

This is the data structure that contains the ID, messages, and severity level of the errors. This is a sample of a structure that is used in the sample app included with this article. It can be found in the Global Data embed point. It includes the override of some Clarion messages, and the addition of some application messages. The first USHORT in the group structure is a constant that informs the object that this structure contains 7 messages. The other USHORT is repeated for each message and contains an equate for the message ID. The standard Clarion messages have the equates in the ABERROR.INC file. The datastructure for the default messages can be found in the ABERROR.TRN file. This structure, by using the same equates (ID's) as the default structure, causes my new messages to be displayed anytime the program encounters one of these situations. To add these messages to the GLOBALERRORS object, you must call the method AddErrors to include/override the messages defined in the data structure above. I placed this call in the window manager init method embed in the Main procedure in my app.

GlobalErrors.AddErrors(AppErrors)

The parameter is the label of the data structure containing my messages.

I also used the AddErrors method to override some messages at the procedure level. These were: the Duplicate Key message shown above, the Delete Record message, and the Add Another Record message. These can be found in the frmFile form procedure in the sample app. Also, when you override the messages at the procedure level, you have to remove them when you leave the procedure. So, I made a call to the RemoveErrors method in the Window Manager Kill embed. This removed my overrides and reinstated the Clarion default messages for any other procedures. In this way, you can make the standard messages custom at the procedure level and create your messages in context with the procedures.

You might notice that there is a BYTE field in each message structure that contains an equate starting with Level:. This is the severity level of the message. There are 5 levels provided by default with the class. Level:Benign, User, Notify, Fatal, and Program. Level:Benign returns Level:Benign and does no further processing. Level:User displays the message and returns Level:Benign or Level:Cancel depending on the user selection of the YES or NO button on the message window. Level:Notify displays the message and returns Level:Benign . Level: Fatal displays the message and halts the program. Level:Program is treated the same as Level:Fatal.

The methods used to handle the processing of the custom messages are:

SetField(‘FieldName’) This method sets the value of the %field expansion macro.

SetFile(‘FileName’) This method sets the value of the %file expansion macro

AddErrors(DataStructure) adds/overrides the messages declared in DataStructure to the error object.

RemoveErrors(DataStructure) removes the messages declared in DataStructure from the error object, and reinstates the original messages if they were overridden by an item in DataStructure.

ThrowMessage(ErrorID,’Message Text’) sets the value of the %message expansion macro and processes the error.

Throw(ErrorID) process the error.

These are the methods used in the sample program provided with this article. Other methods of note are:

ThrowFile(ErrorID,’FileName’) sets the value of the %file expansion macro and process the error.

SetFatality(ErrorID,SeverityLevel) changes the severity of the error specified by ErrorID.

You can look up the other methods used by this object in the Application Handbook included with C4. I have only shown those methods I have used in the sample app along with a couple of others that you might use.

To make things even easier, I created a function in the app to handle the ‘Throw’ methods used to eliminate even the few lines of repetitive code needed to process the errors. The structure of this function is such that it calls the appropriate ‘Throw’ method and the SetField, or SetFile method based on four optional parameters. The prototype of the function is:

```
DoError (<USHORT>,<STRING>,<STRING>,<STRING>)
```

The code in this function processes the proper ‘Throw’ method based on the inclusion of certain parameters.

```
IF NOT OMITTED(2)
  GlobalErrors.SetField(xField)
END
IF NOT OMITTED(3)
  GlobalErrors.SetFile(xFile)
END
IF NOT OMITTED(1)
  IF NOT OMITTED(4)
    GlobalErrors.ThrowMessage(xErrID,xMsg)
  ELSE
    GlobalErrors.Throw(xErrID)
  END
ELSE
  GlobalErrors.Throw(Msg:UnknownError)
END
```

As you can see, I only have these method calls in one place in the app, and each time I need to process an error or message, I just add one line of code:

```
DoError(Param1,Param2,Param3,Param4)
```

By omitting any parameter(s), I can control how and which type of error message is called.

Please play with the sample included here and see how easy it is to add your own custom errors and messages, as well as override the standard clarion errors/messages. Check out my override of the ToDo message. It can even be used as a marketing tool for your application for added functionality.

This is another example where OOP actually simplifies our life, after we learn how it works. I was dragged into the OOP world kicking and screaming when C4 first went Beta, and then I started testing and playing with it. Now, I don’t know how I got along without it. So many things are so much easier to manipulate now. Even Errors and Processing Messages.

If you have any comments or questions about the sample app, or this article, please email me at gcobaugh@ktsoftware.com. I might even answer<g>. Until we meet again, right here in our own little corner of cyberspace, Happy Coding.

Tool Talk: I, Object

By Tom Hebenstreit, Reviews Editor

Hmmmm. Let's see – last week Alan Telford had a nice article on getting what he called Business Objects integrated into his applications, and this week I will be discussing pretty much the same concept using another freely available tool.

Do we see a trend here?

You bet your bippy we do. It is my opinion that many Clarion developers are becoming more and more convinced of the utility of objects, even those who aren't fans (or even users) of ABC. Unfortunately, as soon as they get excited about writing and using their own classes and objects, they run smack-dab into a wall: There are no tools at all in the box to help you write and use your objects (no, I'm not including the source editor as a RAD tool!).

A Little History

A few years back, I was at a DevCon where then TopSpeed CEO Bruce Barrington proclaimed that Borland, with their mantra of objects-objects-objects, was barking up the wrong tree. TopSpeed's superior answer was, as you might guess, templates-templates-templates.

Only a few years later, I find it somewhat ironic that after jumping whole hog into objects-objects-objects themselves, the only way that TopSpeed provided for you to integrate your own object classes into their RAD environment was via the distinctly non-RAD method of hand-coding everything.

Excuse me, but *someone* is missing the point here, and I don't think it's me. (And yes, I know that Clarion is now a SoftVelocity product, but I don't feel that SV should be whacked on the head for this long-standing omission in the Clarion IDE. At least not yet…)

Fortunately for the hand-code-a-phobics among us, a few kind souls have come up with solutions to this dilemma. And how did they do it? By using templates-templates-templates, of course.

Getting Objective

The good people at CapeSoft provided the original solution to the problem over two years ago in the form of a free template set called ObjectWriter, currently at version 1.5. The stated goal for ObjectWriter is to try and make custom classes and objects more visible within the Clarion IDE, as well as taking greater advantage of the IDE and templates to automate much of the tedious detail work that hand-coded classes require.

Does that sound familiar? It should, as that is the basic premise of Alan Telford’s article and templates as well.

By the way, if you haven’t read Alan’s article yet, go ahead and take a look at it now – it will save some time and I won’t have to wade through some of the basic concepts again here (thanks, Alan!). I’ll also be pointing out a few areas where the two template sets use different approaches.

Installing ObjectWriter

The ObjectWriter templates have a professional quality install that pretty much does it all for you. Clear instructions are provided in an HTML file that is easy to view and print.

ObjectWriter stores the INC and CLW class files it generates in your Clarion \LibSrc folder, right along with ABC and any other classes you may have floating around. Classes can be made ABC compliant by flipping a switch, but it is not a requirement. You can use your classes in both ABC and Legacy application, depending on how you code them, of course. By that I mean you cannot create a class that uses ABC methods, and then expect it to work in Legacy (and vice-versa).

In my case, I am using ObjectWriter to add classes to an ABC-based multi-DLL application where I want to both share certain global objects across the EXE and DLLs, and also create local objects as needed.

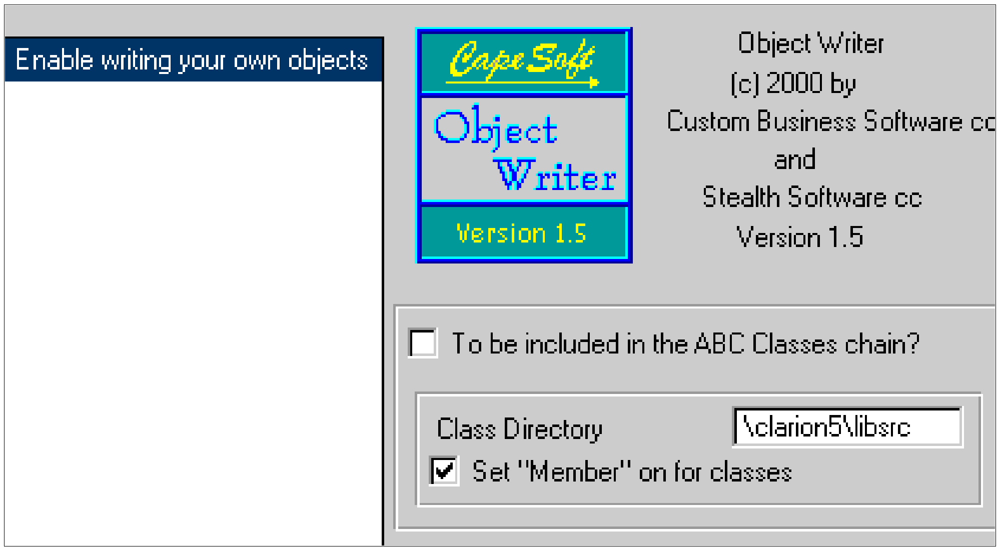
Let’s start at the beginning and see just how hard it is to create a class and share it in multiple applications.

Slowly I turned, Step By Step

The first thing I did was create a separate application where I could store all of my class definitions. This isn’t really necessary, but after some experimentation it just seemed logical to keep all of my classes together.

Within this app, the first thing to do was add a global extension template that activated ObjectWriter.

Figure 1. The ObjectWriter Global extension template



Enable writing your own objects

CapeSoft Object Writer Version 1.5

Object Writer
(c) 2000 by
Custom Business Software cc
and
Stealth Software cc
Version 1.5

☐ To be included in the ABC Classes chain?

Class Directory: \clarion5\libsrc

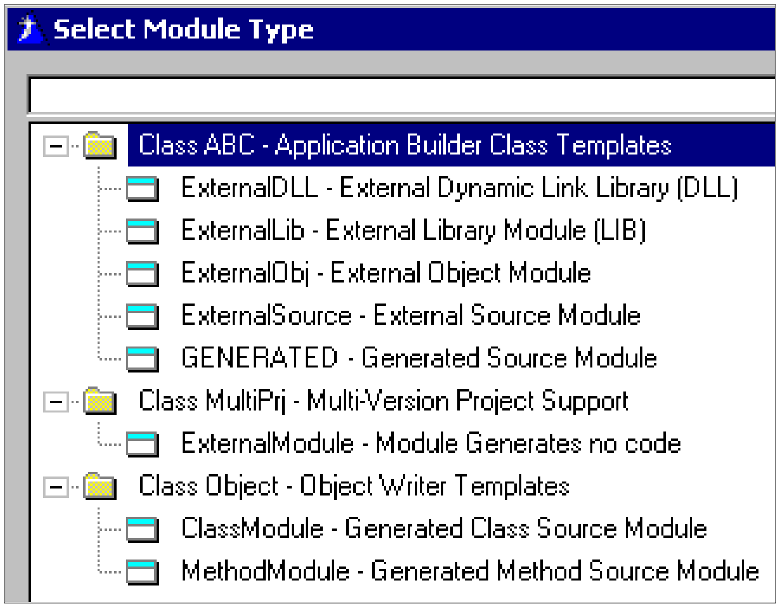
☑ Set "Member" on for classes

As you can see, your choices are pretty simple. In my case I chose not to include my classes within the ABC chain. Why? Because they will contain information exclusive to my application, and if I make them part of the ABC chain they would automatically be included in the dictionary DLL of every other multi-DLL application I write (meaning lots of hassles and compile errors). I also checked the Set Member On button, because I knew these classes were going to require access to and knowledge of the applications data files.

Now, it would have been entirely possible to do what I wanted to do using ABC compliant classes and keeping the Member switch off. For example, I could have abstracted the file handling similar to the way that the ABC `FileManager` does things, but that was also entirely more effort than I wanted to expend in this case. Why build an airplane if you only need to cross the street?

Moving along, the next step was to create two special modules, using new options provided by the templates. The first would be the class include (INC) file, while the second would be the actual code (the CLW) file. To do this I just used the Application | Insert Module menu options. As shown in figure two, there are two new module types provided by ObjectWriter.

Figure 2. The ObjectWriter Module options



Select Module Type

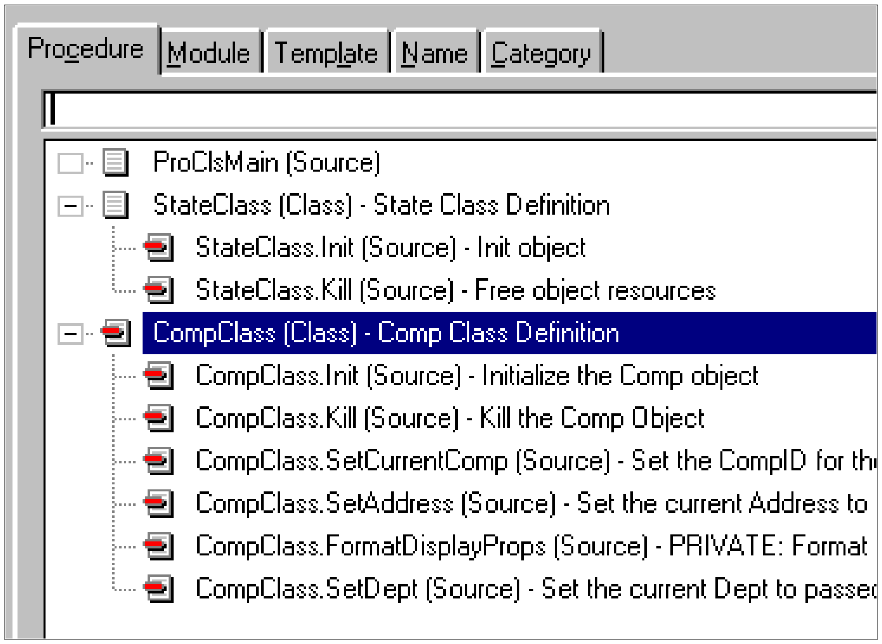
- Class ABC - Application Builder Class Templates
  - ExternalDLL - External Dynamic Link Library (DLL)
  - ExternalLib - External Library Module (LIB)
  - ExternalObj - External Object Module
  - ExternalSource - External Source Module
  - GENERATED - Generated Source Module
- Class MultiPrj - Multi-Version Project Support
  - ExternalModule - Module Generates no code
- Class Object - Object Writer Templates
  - ClassModule - Generated Class Source Module
  - MethodModule - Generated Method Source Module

Creating one of each, I now had a place to put my classes. In the INC module, I added a new procedure called `StateClass` using the ObjectWriter Class Description template. This is really the key template in ObjectWriter, as it is here that you specify your class properties and methods. How? There are two extra buttons on the procedure properties: Properties and Methods. You just type in the properties, specifying the data types and other pertinent information. For Methods, you use the fully qualified name and specify if the method is public, private or protected; and also whether it is normal or virtual. When I say fully qualified, I mean that you have to include the class name. For example, the procedure name for the Init method of my `StateClass` was named `StateClass.Init`.

Once you have entered your properties and method names and saved the Class definition procedure, your methods appear on the application procedure tree as a ToDo just like any other new procedure. When you double click on them, you can choose the procedure type and then enter the prototypes, etc. just like you normally would. I pretty much always choose the Source procedure, but CapeSoft says that they have had success using the Window Procedure as well – and that can mean no hand-coding windows and accept loop processing.

So, after adding a couple of simple classes to my application, it ended up looking like this:

Figure 3. App procedure tree using ObjectWriter



ProClsMain (Source)
StateClass (Class) - State Class Definition
StateClass.Init (Source) - Init object
StateClass.Kill (Source) - Free object resources
CompClass (Class) - Comp Class Definition
CompClass.Init (Source) - Initialize the Comp object
CompClass.Kill (Source) - Kill the Comp Object
CompClass.SetCurrentComp (Source) - Set the CompID for th
CompClass.SetAddress (Source) - Set the current Address to
CompClass.FormatDisplayProps (Source) - PRIVATE: Format
CompClass.SetDept (Source) - Set the current Dept to passe

Neat, clean and visually clear – I like it. Click on the Class definition to change properties; click on the method to write the code. ObjectWriter handles everything else. It generates the INC and CLW files, handles setting all of the required mode flags, and basically makes it reasonable to work with your own classes within the Clarion IDE.

If you read Alan's article, you'll see one major difference between his templates and ObjectWriter: His templates place all method code within one procedure, while ObjectWriter normally places each method within its own AppGen procedure.

Which way is better? That depends on the types of classes you write. If you have myriads of tiny methods, the OW method may become cumbersome (but no more so than any large app).

Sharing The Wealth

Ok, you have these fabulous classes all ready to go now – so how do you use them in the other DLLs or an application EXE?

Elementary, my dear Watson. ObjectWriter has another global extension that you add to your other apps called, amazingly enough, "Use other objects in this application." There, all you do is specify the name of the class INC file. For example, I had named my class INC and CLW files ObjCls32. By entering that in the list of objects I wanted to use, I now had complete freedom to use the class in my application. To illustrate, I could declare an instance of the StateClass you saw in Figure 3:

```
MyStates StateClass ! From ObjectWriter app/DLL
```

I could then use it like this:

```
MyStates.Init
Message('MyState class||Mystates.RecCount: ' |
  & MyStates.RecCount & ' on ' |
  & format(MyStates.OpenDate,@D17),'**TEST**',ICON:HAND)
```

In this trivial example, the Init method opened and read a file, also setting some properties such as the number of records in the file and the date opened.

Wrapping It All Up

I have only skimmed over many of the more advanced features of ObjectWriter, but I hope you now have a decent feel for how simple it is to use.

Of course, neither ObjectWriter or Alan's templates address the *real* problem, and that is that this type of functionality should be built into the IDE.

Having two free solutions at hand does take a lot of the pain out of the process though, and they both prove that it wouldn't be *that* hard for SV to implement something natively within the IDE.

By the way, both templates are also the product of real-world situations. Bruce Johnson of CapeSoft has told me that they use ObjectWriter to help build many of their other products, and Alan's was obviously oriented around practical goals.

I haven't used Alan's templates yet, but I *can* vouch for how much easier ObjectWriter has made my life in that multi-DLL application I am building. I highly recommend that you check out both tools – you certainly can't complain about the price.

So… thanks guys! The Clarion community owes you a big round of applause!

ObjectWriter can be downloaded free of charge from http://www.capesoft.com.

A longtime Clarion user, Tom Hebenstreit is an admitted tool junkie who refuses to go straight and code without his arsenal of third party products. During those rare moments when he isn't either using or writing about Clarion, he indulges his twin passions for blues and beer by performing around Southern California in a variety of totally-obscure-but-famous-any-day-now rock and blues bands.

Clarion MAGAZINE

Tool Talk: I, Object
(Aug 22,2000)

Five Rules for Managing Complexity
(Aug 22,2000)

August 2000 News
(Aug 22,2000)

Five Rules for Managing Complexity

by Tom Ruby

Part 1

Programs are very complex things because users demand that they solve complex problems. You know to be wary when the customer says, "All it needs to do…" because there is always more to it than that. Your task is to analyze that complexity and produce a program that deals with it, leaving the user to think, "All I need to do is…"

This is a pretty tall order, and the last thing we need to do is add more complexity to the problem ourselves. Within our programs, dictionaries and embeds, we want our work to be as simple and straight forward as we can make it so we can concentrate on the complexity of the problem, not on the complexity of our solution.

So first, here is a guideline. It's not a hard and fast rule you must follow, but an example of a principle which can save you some hair.

> **Guideline 1**
> Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 1 suggests a strategy, or overall guide, which might be, "Make up a set of rules and stick to them." Consider a rule I follow when making dictionaries,, "Make field names descriptive and don't abbreviate them." It is always a temptation to name a field something like `Date`. Hmmmm. There already is a field in this table called `Date`, and it is the date the record was entered. So I call the new field `DateDue`. But now I have an extra piece of information to keep in my head, and this information is about the solution to the problem, not about the problem, so it adds to the complexity of the task needlessly. This bit of information is, "The date entered is `Date`, while the due date is `DateDue`."

Keeping Guideline 1 in mind, I make the other date field more descriptive and change it to `EntDate`. Oops. This is an abbreviation. Why is that bad? Because it adds another piece of information to keep track of. I have to remember that I abbreviated Date Entered as `EntDate`. So I fix it by making it `EnteredDate`.

Now I feel much better, but before long, I have an annoying compile error because I typed `DueDate` instead of `DateDue`. Ok, so it isn't a big mistake, but how many times throughout the project do you want to get a compile error because of this mistake? So let's make an arbitrary rule, "The type of the field comes before the use of the field." It could easily be the other way around and it will work fine. In fact, my own rule for naming fields is the opposite. It doesn't matter - the purpose of the rule is just to remove unnecessary complexity.

The rule also makes it clear that the tax amount field is `AmountTax` and the tax percentage field is `PercentTax`. Already this rule has saved four scraps of unneeded information about the solution, and will probably save more later on.

Yes, I know Clarion has that handy fieldbox, plus Data Modeler and the View Dictionary feature. I love them all and use them heavily, but which is better, looking up a field name or just knowing it instinctively? There's a lot these tools can help you with, but you don't need to be using them to solve problems you could have designed out at the start.

"Naw, let's not change it 'cause I'll have to reedit all the embeds in that procedure." Again, how many times throughout the project do you want to make the mistake and get an annoying compile error? Change it now while the fix is easy to make. This brings up Guideline 2, which is more of an observation than a rule:

> **Guideline 2**
> It is a lot less painful to fix a painful mistake now than it will be later on.

As I explained to my son Ethan a few days ago, "It hurts to get a nasty splinter out, but it will hurt more to get it out next week. Do you really want to put up with it till then?" He opted to let me take the splinter out. So I'll bite the bullet and edit the embeds in the procedure. My, the Embeditor is handy for things like this.

Let's talk about abbreviations a bit. Would you make a field name `AmountFederalInsuranceContributionsAct`, or `AmountFICA`? FICA is indeed an abbreviation for Federal insurance Contributions Act, but FICA is also the name of a fairly complex, not to mention wordy, concept. I'd consider FICA a name and call the field `AmountFICA`, but I wouldn't abbreviate it as `FICAAmt`.

I could go on with examples all day, but I won't. Take it from twentymumble years of sometimes frustrating experience: Stick to your rules!

To recap:

> **Guideline 1**
> Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.
>
> **Guideline 2**
> It is a lot less painful to fix a painful mistake now than it will be later on.

The First Rule

And now, without further ado, here is Rule Number 1:

> **Rule Number 1**
> Eliminate repeating fields.

People often paraphrase this rule by saying, "Arrays are bad," but this tells only part of the story. I have actually seen people eliminate the bad array by defining their data like this:

```
EmployeeRate1
EmployeeRate2
EmployeeRate3
EmpRate4      ! must have got tired of typing
EpmRate5      ! didn’t proof read
EmpRt6
```

and so on.

And this changes fairly logical and easy to maintain code like this:

```
LOOP I = 1 TO 6 BY 1
   YadaYada
   EmployeeRate[I] Yada
   Yada
   Yada(EmployeeRate[I])
   Yada
END
```

Into a mess like:

```
Yada Yada
EmployeeRate1 Yada Yada
Yada Yada Yada Yada
Yada EmployeeRate1 Yada Yada Yada
Yada Yada
EmployeeRate2 Yada Yada Yada
Yada Yada
Yada EmployeeRate2 Yada Yada Yada
Yada Yada Yada
(lots more nonsense)
Yada
Yada EmpRate4 Yada
Yada Yada Yada Yada
```

So what’s so bad about this? You can easily build it using cut and paste or an editor macro, right? Well, it turns out that the second `Yada` should be

```
Gobbledegook EmpPlanA
Gobbledegood EmployeePlaB
```

and so on for all umpteen bunches of code! Somebody has a long day of tedious editing to do. Do you think they’re not going to goof somewhere? What if the you code `EmployeePlanC` where you should have put `EmployeePlaB`? Just try figuring that bug out while the client is looking at screwed up calculations! Then you have to read through that 24 page embed, line by line, looking at these field names which all look alike but are subtly different. Are you sure you’re not going to flub up while fixing them?

So getting rid of the array by replacing it with bunches of fields is not a good idea, but what’s wrong with the array concept itself? The array causes a big limitation in the functionality of the program, and it adds a bit of complexity you can do without.

When you put an array in your data, you place a limitation on your program's usefulness. I call this "The Extension Cord Effect." Ever notice how often an extension cord is just a little too short? You could just always buy nine foot extension cords instead of six foot cords, but you know, the longer cords somehow just need to be a couple inches longer. Of course, you could always buy 100 ft outdoor cords, but then you'd have all this orange cord laying around, tripping up the kids and tangling the vacuum cleaner.

The same thing happens to an array in your data design: You've allowed for eight employee deductions, but doggone it, you get a phone call interrupting your dinner because one of the customers needs 24, so you have to dig out the source code, make a change and update their database replacing the eight element array with a larger one. Then *everybody* gets space for 24, while most use only three, and it works fine until somebody needs 25. Why in the world would they need 25? I don't know, but if the user thinks they need 25, the program shouldn't tell them they can't have 25. And you did remember to make *all* occurrences of `LOOP I = 1 TO 24 BY 1` into `LOOP I = 1 TO 25 BY 1`, didn't you? Everywhere? Are you sure? How sure are you?

A repeating field, be it an array or multiple fields, adds another extra bit of complexity to the project, and you certainly don't want to be making it more complex than it needs to be. You need a way to tell if the array element is used and skip over those that aren't. You need to put the code to do this *everywhere* you use the data out of the repeating fields, and you have to remember, yet again, how you coded the unused fields. You probably also need to move them around when the user deletes one because users always like the blanks at the bottom.

What to do? Put the repeating fields in a separate table, one field, or set of fields, per record, along with the employee's identifier so you know which records belong to which employee. For this example, the `EmployeePlan` table would contain three fields: the employee identifier, the rate field and the plan field. Now the code looks like this:

```
PLA:Employee = employee identifier
SET(EmployeePlanKey,EmployeePlanKey)
LOOP
  IF Access:EmployeePlan.Next() <> LEVEL:Benign OR |
    PLA:Employee <> employeeidentifier THEN BREAK END
        Yada Yada Yada Yada Yada Yada
        PLA:EmployeeRate Yada Yada
   END
END
```

Oops, remember to change the second `Yada` to

`Gobbledegook EMP:Plan`

You don't have to worry about skipping unused records, because there aren't any unused records and the simple loop works for as many plans as are or aren't used. You also don't have to worry about how many records there are. The loop will just process all of them whether you have 200,000 entries, one entry or zero entries.

"But my users will hate another browse and form," you say. I don't blame them. There are better ways. The most obvious solution is Clarion's Edit-In-Place feature. You put a browse for the plan records on the employee form and range limit it to the employee identifier field. Then you add the update buttons and check Use Edit-In-Place. In the example application, I changed the class of the second column and added a single line of embed code to make it a drop list.

If your users, or your clients, are really stuck on the idea that there are a certain number of these repeating fields, you can fake them out on the form while still getting rid of the repeating fields. Remember, user interface design is in large part psychology. The trick is to make a set of fields on the form and write these to the child table. In the `FRMFakeOut` procedure, I used an intermediate queue to allow the user to "roll" the list through the edit fields. It acts like 10 fields and the users don't realize that you made your life easier, besides getting their program done faster and with less grief. Is the client going to freak out over the scroll up and down buttons? Hide them, but leave some hidden configuration parameter to unhide them with `PROP:Hide`, just in case.

Problems With Reports

Making the number of fields potentially unlimited can present reporting challenges. "But only four columns fit across the paper." Now that's a problem. Paper is just 8 ½ inches wide in the US. The rest of you have it worse with 210 millimeter wide paper. With any report, you have to ask, "What does the user want to know by looking at this report," and secondly, "What is the user going to do with this piece of paper after looking at it."

When the computer was the great machine hidden away in a special room where only the white coated priests could enter, reports were of extreme importance. That was the only way information could get from the computer to the users. These days, we put computers everywhere. The high school kid at Wal*Mart who makes minimum wage works in front of a computer. We don't mess with those 3 ½ inch thick books of 14 inch greenbar paper anymore because it is much more convenient to put the information on a screen than to look it up in that huge printout. Ask your users what they are going to do with that 248 page "Master Inventory Report." I once had a client demand this very important report, but it was never used. You see, the Inventory browse was much more useful to his employees.

That aside, what do the users want the report for? I'm guessing the problem is a report that shows a line for each employee and a column for each something else. It is more important to find out what the user wants to know from the report than how they want the report formatted. Perhaps only the four latest records for each employee are significant, or four different totals of all the records for each employee would be more interesting. Perhaps the only reason they want it is because they always had it. Are they looking for something specific in the grid, like a number that is much greater than or less than the others? Or are they looking for an unusually high column total?

I failed to ask the "purpose" question once and ran two mainframe jobs, each producing a four inch thick stack of 14 inch greenbar paper, to satisfy a coworker's request. Turns out all she wanted to know was how many records there were. I could have told her that immediately when she first asked for the report rather than four days and eight inches of paper later!

Say that after the analysis you're still faced with a report with only four columns for each employee. Do you need to restrict the data design for this report, or does the report itself exhibit the "extension cord problem?" That is, what happens when, and not if, somebody needs the seventh column? Rather than restricting your data design based on this report, you need to design the report to deal with the inevitable.

What Good *Are* Arrays In Tables?

"How can it be so bad if Clarion allows it?" Clarion allows you to put arrays in data tables because it has to. There are three reasons Clarion has to allow it.

The main reason is if they didn't, some numskull will complain loudly in a public place that Clarion is totally unusable because it lacks this important feature.

The most important reason is that we always have to deal with data from other programs. That's the whole reason we have drivers like dBaseIII. Is there an advantage in storing some table in a dBaseIII file rather than a TPS, Btrieve or SQL table? No. The reason we use the dBase driver is that some other program requires its input or output to be in dBaseIII files. That's why Clarion allows arrays in the data files, because the nit who wrote the other program you have to interface with didn't know any better.

I said there are three reasons, didn't I? The third reason is they have to support people who are too busy with problematic projects to read an article like this about how to design the problems out of a project.

Sometimes you have what looks like a repeating field but isn't. Consider a table which represents 1099 forms. There are a lot of numbers on the 1099, and if you're looking at an old program somebody else wrote, they might have stored these in an array. To me, fields named `DollarsRents`, `DollarsRoyalties`, `DollarsOther` and so on are more meaningful than `AMT[1], AMT[2]` and suchlike. These 11 fields are not repeating fields, so don't feel bad about putting them in the 1099 table.

To recap, here are the two Guidelines and Rule Number 1:

> **Guideline 1**
> Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.
>
> **Guideline 2**
> It is a lot less painful to fix a painful mistake now than it will be later on.
>
> **Rule Number 1**
> Eliminate repeating fields.

Next time, Rule Number 2 and some more guidelines.

Download the example application

Tom Ruby, who is no relation to the man who shot Lee Harvey Oswald, is an independent contractor living in the middle of a hayfield in Central Illinois with his wife Susan and two red-headed sons, Caleb and Ethan. He has been using Clarion for Windows since the summer of '95. Before that, he was a "TopSpeeder" using Modula II, so he has never used the DOS versions of Clarion.

Learning To Write A Business Object

by Alan Telford

Clarion is defined as a business programming language. Its main strength is having a data dictionary and code-generating templates which use that dictionary to generate browses, forms and reports, with very little hand coding required. Clarion is also an OO language. However Clarion has not made writing your own objects as easy as writing your own browse or form.

There are many examples of OOP around. Most of them are utility objects (resizing a window) or programmer objects (error class, file manager). But few if any are what I would call business objects. And being a business programmer I want to write business objects – objects that know about my data files. These objects would handle payroll procedures, manipulate invoices, deliveries, etc. In this article I will show you one way to go about it.

I want to add a business object like I add a browse. Select a template, add some files, add some embed code – then click the lightning bolt and start using it. This kind of object knows about the data files that are inside the dictionary. It's not a generic object and won't go in the Clarion5\Libsrc folder, but will be stored inside my application. The disadvantage of course is that the object is tied to my specific application, but the advantages are:

- The ABC templates have automatically declared the dictionary file, and given it a FileManager and RelationManager object, and initialized these objects
- The populate field listbox is available enabling the programmer to double click on a field or key saving typing and possible errors that may occur.
- I can simply back up the APP and DCT and know I've got the lot without having to hunt for some separate class objects.

Public Holidays

As one example of a business object I need to know whether a date is a public holiday or not. The only information I need to know about public holidays is a date and a name. So add this file to your dictionary. Since it's inside the dictionary, simply choose browse wizard to add a browse and form for updating.

```
Holidays FILE,DRIVER('TOPSPEED'),↵
                PRE(HOL),BINDABLE,CREATE,THREAD
K_Date     KEY(HOL:Date),NOCASE,OPT,PRIMARY
Record     RECORD,PRE()
Date         DATE
Name         STRING(20)
           END
         END
```

When calculating a payroll an employee will get paid with a different contract or a higher hourly rate on public holidays. If I was hand-coding this I could write:

```
HOL:Date = Date
IF Access:Holidays.fetch(HOL:K_Date) = LEVEL:Benign
  ! employee gets paid more
END
```

But this makes my payroll procedure aware of how holidays are stored. A good object should hide more information that it reveals. And if this test is repeated 5-10 times throughout my payroll calculation procedure there will be 5-10 copies of the code. I don’t like duplicating code. One easy solution to code duplication would be to turn this into a procedure call:

```
IF IsPublicHoliday( date )
  ! employee gets paid more
END
```

But the goal is to learn about writing objects. What I’d really like to have is an object called *Holiday* which knows about all the public holidays. Then I could just ask this object if a holiday exists on a date. E.g.

```
IF Holiday.exists( date)
  ! employee gets paid more
END
```

This code certainly looks like OOP and is self commenting. So how do I write this holiday object?

Making A Local Class

The most basic way to start writing OOP is right inside one of your procedures. Declare the class inside the *Local Data* embed point, and then write the class methods inside the *Local Procedures* embed point

Step 1. Inside embed point *Local Data – Other Declarations* type the following:

```
HolidayClass  Class,Type
Exists        Procedure(Long p:Date),Byte
              end
Holiday       HolidayClass
```

The first 3 lines declare a new type of class called *HolidayClass*, and then the last line declares an object called *Holiday* of this type.

Step 2. Write the actual method. This goes in the *Local Procedures* embed point.

```
HolidayClass.Exists Procedure(Long p:Date)
  CODE
  HOL:Date = p:Date
  RETURN |
   Choose(Access:Holiday.fetch(HOL:K_Date) = LEVEL:Benign)
```

Adding More Methods

There are many more questions you could ask of a *Holiday* object.

- How many public holidays fall in the payroll period?
- What is the name of this public holiday?
- What's the next public holiday on or after a given date?

Of course my payroll procedure shouldn't calculate the information itself. It should ask the questions directly of the *Holiday* object. For the above examples this could be written:

```
! how many public holidays fall in the payroll period?
Message('NumberOfHolidays = ' |
  & Holiday.count(startdate, enddate))
! what is the name of this public holiday
Message('Name of holiday is '&holiday.getName( Date))
! what's the next public holiday on or after today.
Holiday.set( today())
NextHoliday = Holiday.next()
```

The changes to the class declaration are:

```
HolidayClass  Class,Type
Exists        Procedure(Long p:Date),Byte
count         Procedure(Long p:startdate, Long p:enddate)
getName       Procedure(Long  p:Date), String
next          Procedure,Long
set           Procedure(Long  p:Date)
              end
Holiday       HolidayClass
```

And the new class methods for *Local Procedures* are:

```
HolidayClass.count Procedure(Long p:StartDate, ↵
   Long p:EndDate)
! Count the number of holidays in date range
! p: StartDate to p: EndDate
HolidayCnt Long,auto
  CODE
  HolidayCnt = 0
  HOL:Date = p:Start
  SET(HOL:K_Date, HOL:K_Date)
  loop until Access:Holidays.next() OR HOL:Date > p:End
    HolidayCnt += 1
  END
  RETURN HolidayCnt
HolidayClass.getName Procedure(Long p:Date)
! Return the name of the public holiday
! on the given date
  CODE
  HOL:Date = p:Date
  Access:Holidays.fetch(HOL:K_Date)
  Return HOL:Name
HolidayClass.set Procedure(Long p:Date)
! Position holiday file at specified date
  CODE
  CLEAR(Holidays)
  HOL:Date = p:Date
  set(HOL:K_Date, HOL:K_Date)
HolidayClass.next Procedure
! Find the next holiday
  CODE
  RETURN Choose(Access:Holidays.next() = |
    LEVEL:Benign, HOL:Date, 0)
```

So have I really achieved anything? Yes. The *holiday* object is taking responsibility for everything to do with public holidays. The payroll procedure is ignorant of how the holidays are stored. When it needs information it has to ask the *holiday* object As long as I keep the class declaration the same, I can rewrite the holiday code anyway I want without breaking my payroll procedure.

TIP: The best way to learn OOP is to start writing OOP. So the next time you need to embed some complex code for a procedure, instead of doing it the old way and writing a ROUTINE why not write a local CLASS instead. All the benefits of OOP will still apply just on a smaller level. This is a very safe way to learn writing business objects.

Making The Holiday Object Accessible Throughout The Application

There is a major shortcoming with the above approach. The object can only be used in that one procedure. As soon as you try using the *Holiday* object inside another procedure you will get an error message. Objects have scope just like data. There are a number of ways to overcome this:

- To make the *Holiday* object accessible at the global level you could move the class declaration from *Local Data* embed to *Global Data* embed point, and move the class definitions from *Local Procedures* to *Program Procedures*. This is easy to do but clutters your global embed points.

- To make the *Holiday* object accessible at the module level you could move the class declaration from *Local Data* embed to *Module Data Section* embed point, and move the class definitions from *Local Procedures* to *Module Data Section*. The Holiday object would be accessible from any procedure which is in the same module. But what if I want to access from another module?
- The most flexible way to make an object available from any procedure is to copy the ABC template approach. (Right click on any ABC window and have a look). All the include files are declared at the Module level before the procedure name. All the local objects (Toolbar, Window, Browses) are declared as local data. This is a flexible approach but it requires the programmer to maintain a separate INC file, and then include the INC file at module level

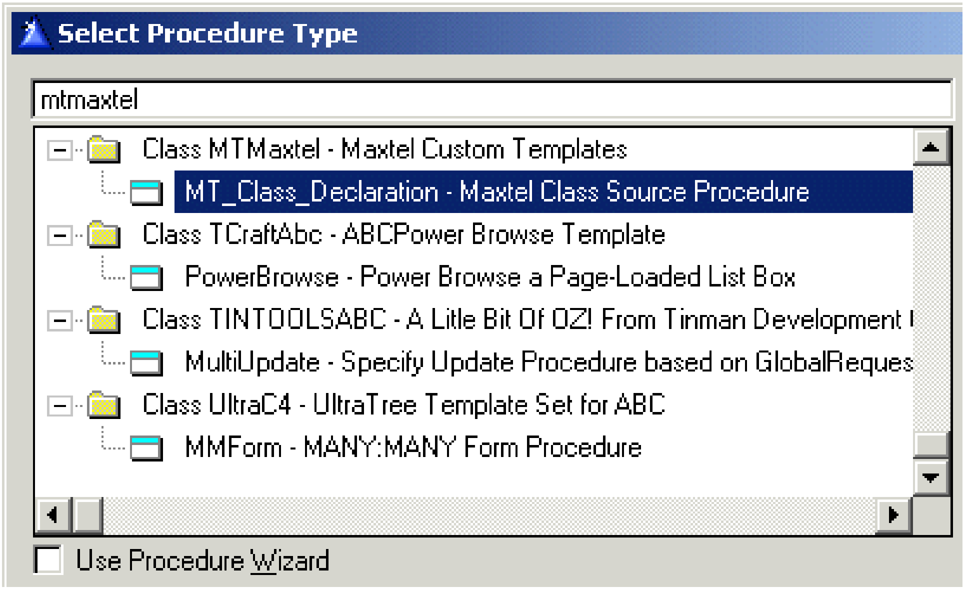
Maintaining the INC file

Clarion can generate source code, so why not let it maintain the INC file?

Included in the attached ZIP file as a template which allows you to write both the class declaration and definition inside your application. Unzip into your tempate directory (Clarion5\Template) and then register the template (MTMaxtel.tpl).

Add a new procedure, give the name `Holiday_Class` and then instead of Browse choose `MT_Class_Declaration`.

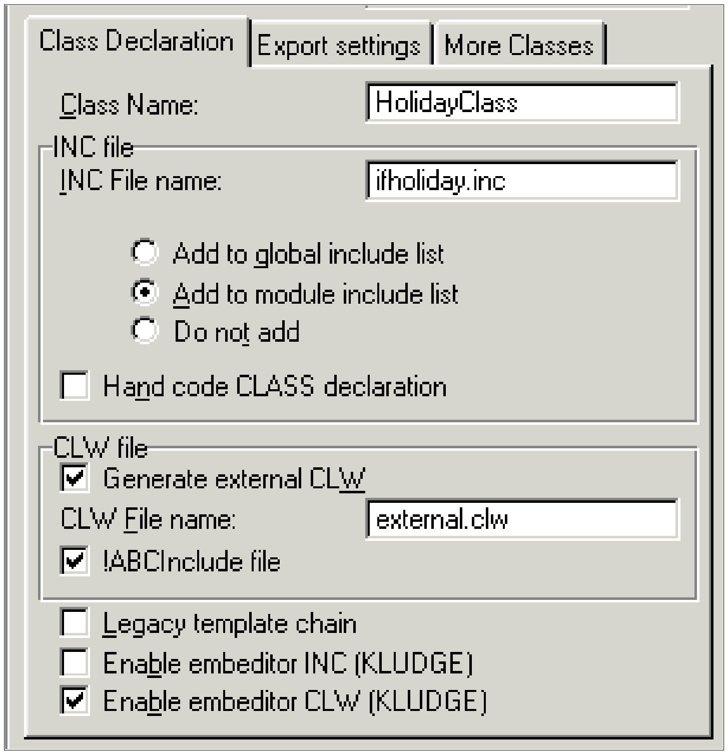
Figure 1. MT_Class_Declaration template


Select Procedure Type
mtmaxtel
Class MTMaxtel - Maxtel Custom Templates
MT_Class_Declaration - Maxtel Class Source Procedure
Class TCraftAbc - ABCPower Browse Template
PowerBrowse - Power Browse a Page-Loaded List Box
Class TINTOOLSABC - A Litle Bit Of OZ! From Tinman Development
MultiUpdate - Specify Update Procedure based on GlobalReques
Class UltraC4 - UltraTree Template Set for ABC
MMForm - MANY:MANY Form Procedure
Use Procedure Wizard

Under Class name enter `HolidayClass`

For INC file name enter `ifholiday.inc`

Figure 2. MT_Class_Declaration settings


Class Declaration | Export settings | More Classes

Class Name: HolidayClass

INC file

INC File name: ifholiday.inc

- ( ) Add to global include list
- (•) Add to module include list
- ( ) Do not add

☐ Hand code CLASS declaration

CLW file

☑ Generate external CLW

CLW File name: external.clw

☑ !ABCInclude file

☐ Legacy template chain

☐ Enable embeditor INC (KLUDGE)

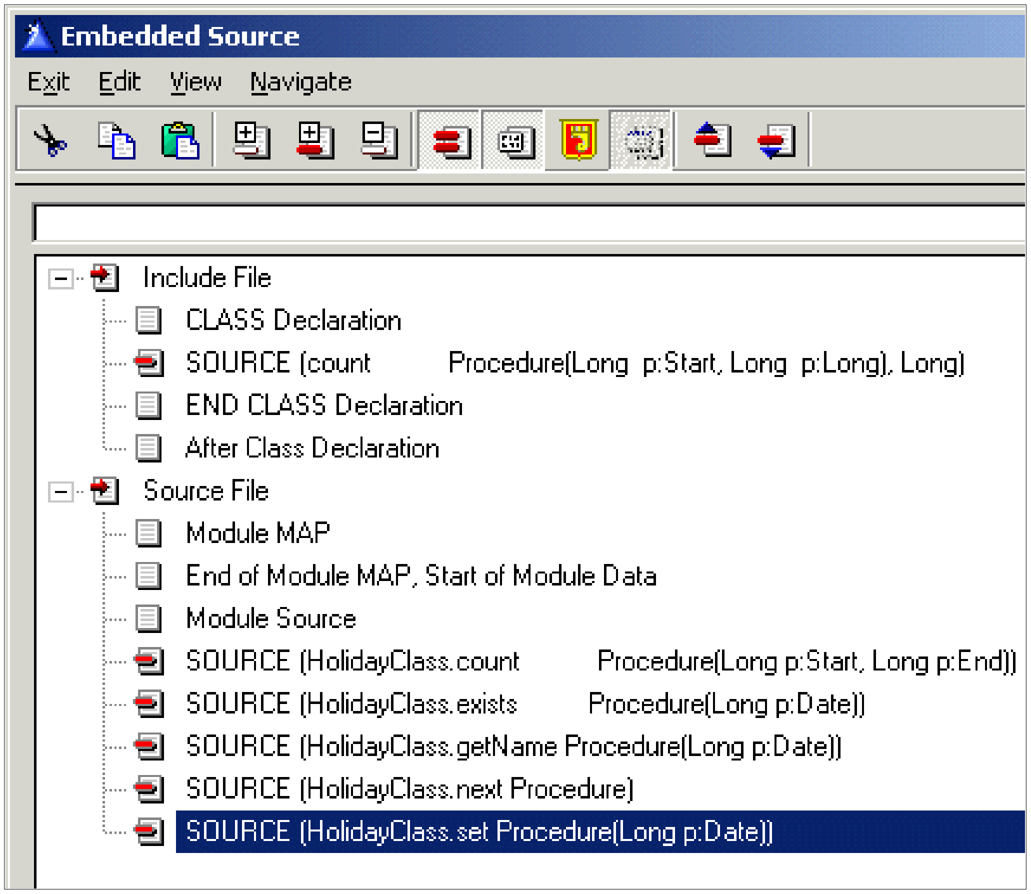
☑ Enable embeditor CLW (KLUDGE)

This template has two main embed points.

- Include File for writing the class declaration.
- Source File for writing the class methods.

So copy the class declaration from your Local Data embed point into the Include File embed point, and copy the class methods from Local Procedures into Source File. Generate your application and you will see the include file ifholiday.inc automatically appear.

Figure 3. Source Embed points



```
Include File
    CLASS Declaration
    SOURCE (count         Procedure(Long  p:Start, Long  p:Long), Long)
    END CLASS Declaration
    After Class Declaration
Source File
    Module MAP
    End of Module MAP, Start of Module Data
    Module Source
    SOURCE (HolidayClass.count         Procedure(Long p:Start, Long p:End))
    SOURCE (HolidayClass.exists        Procedure(Long p:Date))
    SOURCE (HolidayClass.getName Procedure(Long p:Date))
    SOURCE (HolidayClass.next Procedure)
    SOURCE (HolidayClass.set Procedure(Long p:Date))
```

This template has a lot of additional functionality. Please experiment with it. On the first tab are options to generate an external CLW file which could even be placed inside the Clarion5\Libsrc folder as an ABC-compliant class. The second tab has options to export the class from the DLL. On the third tab you can enter additional classes to include in this INC file. Each additional class creates two extra embed points – one embed for the class declaration, and one embed for the class methods.

This allows you to use an APP to maintain all your class objects, with all the advantages this brings.

Using The Business Object

So now I have a template which allows me to easily create classes inside an APP file. To use one of these objects I still need to declare the object in local data, and declare the include file in the module. Sounds like a good job for another template.

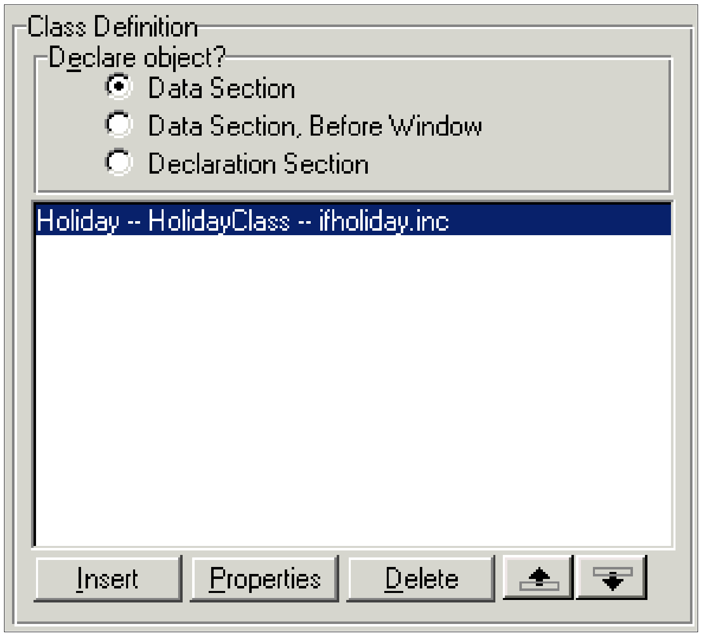
Add the *MTIncludeObject* extension template to the procedure.

For ABC templates, always choose declare object in "Data Section."

"Data Section, Before Window" is useful for legacy Browse/Form templates.

"Declaration Section" is the embed point for Report templates.

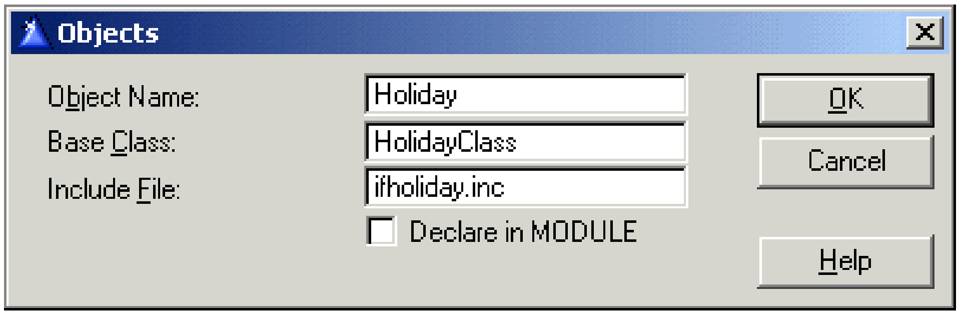
Figure 4. MTIncludeObject template


Class Definition
Declare object?
(● Data Section) ○ Data Section, Before Window ○ Declaration Section

Holiday -- HolidayClass -- ifholiday.inc

Then simply add a list of objects (Name, Class, Include File) into the list.

Figure 5. MTIncludeObject settings



Object Name: Holiday
Base Class: HolidayClass
Include File: ifholiday.inc
☐ Declare in MODULE

The Include File will automatically be placed at the module data level.

The objects will be declared as local data (unless "Declare in MODULE" is selected)

Summary

Clarion's approach of writing classes inside Clarion5\Libsrc folder is appropriate for utility classes which are generic to all applications. But if you want to write a business object which has direct knowledge of the data files inside your dictionary, then choose to write the objects inside your application.

A good way to start is by writing local classes directly inside a procedure. This can replace a number of routines, or be used to prototype a future business object you're starting to write.

Then if you want to make your business object accessible throughout your application, use the two templates provided with this article:

`MT_Class_Declaration` which allows you to write both the declaration and the methods for a class inside your application. This means the object is instantly aware of the files in your dictionary. All the benefits of normal embed code writing are available to you now. Note: ensure there is only one `MT_Class_Declaration` per module inside your application.

`MTIncludeObject` allows you to easily declare your business objects for use inside any other procedures in your application.

Business OOP programming is not as hard as it sounds. This may not be the way that Topspeed (SoftVelocity) imagined it, but it sure does work.

Note: All that I've done applies to Clarion LEGACY templates just as well as ABC templates. You can write your business objects in either type of app.

Download the source

Alan Telford has been programming in Clarion since 1994. He is the Chief Software Developer at Maxtel Software Ltd, a New Zealand software company specializing in writing back office computer solutions for McDonald's Family Restaurants and other similar markets.

Main Menu | Log In | Subscribe | Links | FAQ | Advertising

Reborn Free CLARION online

published by CoveComm Inc.

Search

Main Page

COL Archive

Log In
Subscribe
Renewals

Frequently Asked Questions

Site Index
Article Index
Author Index
Links To Other Sites

Downloads
Open Source Project
Issues in PDF Format
Free Software

Advertising

Contact Us

Clarion MAGAZINE

Legacy to ABC: There is Another Way!

by Simon Brewer

Part 3

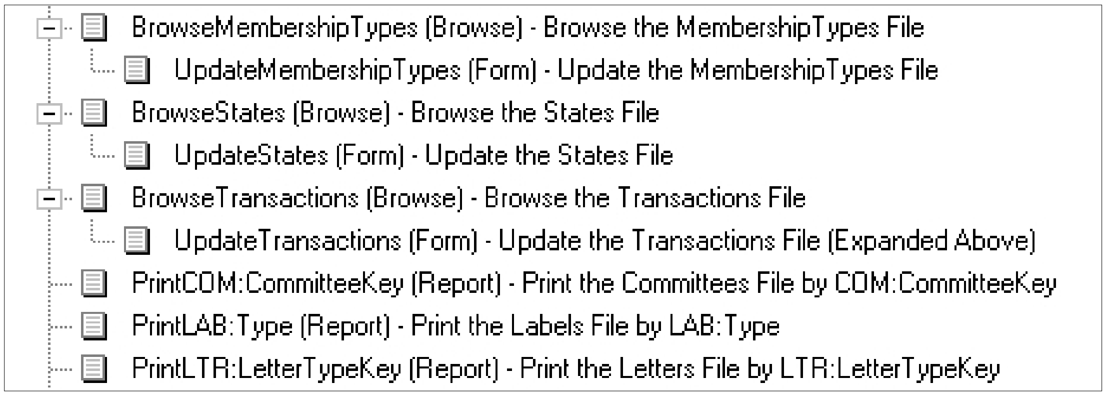
Today is payday. In Part 1 and Part 2 of this article, I've demonstrated - and you've programmed - a hybrid Legacy/ABC app. Today you'll extend that app by converting Legacy procedures to ABC, but just the ones you want to. You'll see that it can be done in as small or as large chunks as you wish with nearly ultimate safety and recoverability.

The Golden Rule – Again!

Before I begin, I must recap the golden rule from Part 2 of this story: ABC procedures must call ABC procedures; they must not call Legacy procedures. That's an important rule in deciding exactly which procedures to convert.

Take a look at Figure 19. It's a snippet of the Application Tree from the Legacy app you created way back in Part 1. I chose this particular section because it has a variety of procedures, including reports. You should be able to locate this section in your app.

Figure 19. Excerpt from Wizard-created Legacy Club Manager application.



```
BrowseMembershipTypes (Browse) - Browse the MembershipTypes File
    UpdateMembershipTypes (Form) - Update the MembershipTypes File
BrowseStates (Browse) - Browse the States File
    UpdateStates (Form) - Update the States File
BrowseTransactions (Browse) - Browse the Transactions File
    UpdateTransactions (Form) - Update the Transactions File (Expanded Above)
PrintCOM:CommitteeKey (Report) - Print the Committees File by COM:CommitteeKey
PrintLAB:Type (Report) - Print the Labels File by LAB:Type
PrintLTR:LetterTypeKey (Report) - Print the Letters File by LTR:LetterTypeKey
```

Arbitrarily, I've decided you'll convert the `BrowseStates` procedure to ABC, which, applying the golden rule, means the `UpdateStates` procedure must also be converted.

Furthermore, the `UpdateMembershipTypes` and `PrintCOM:CommitteeKey` procedures seem ripe for conversion too.

In all honesty these procedures have been chosen entirely at random. It's worth pointing out that you could convert just one procedure if you wanted, or any number. I suggest you keep the number small and manageable. In fact, I recommend that you keep to "sub-trees" or sub-applications to help maintain your own sanity.

It's worth noting that a fair amount of the old *80:20 Rule* applies to most applications; that is, about 20% of your application requires about 80% of the effort. Therefore, for any conversions you do using this hybrid method, it's good to pick all of the simple procedures, such as those with few or no embeds that use standard templates, and convert them first. Pretty soon you'll have an 80% complete ABC application, and you can clearly see the *harder* procedures nicely separated from the others. That'll make decisions such as whether to convert or re-write those remaining procedures a lot easier. Suddenly they won't look nearly as hard!

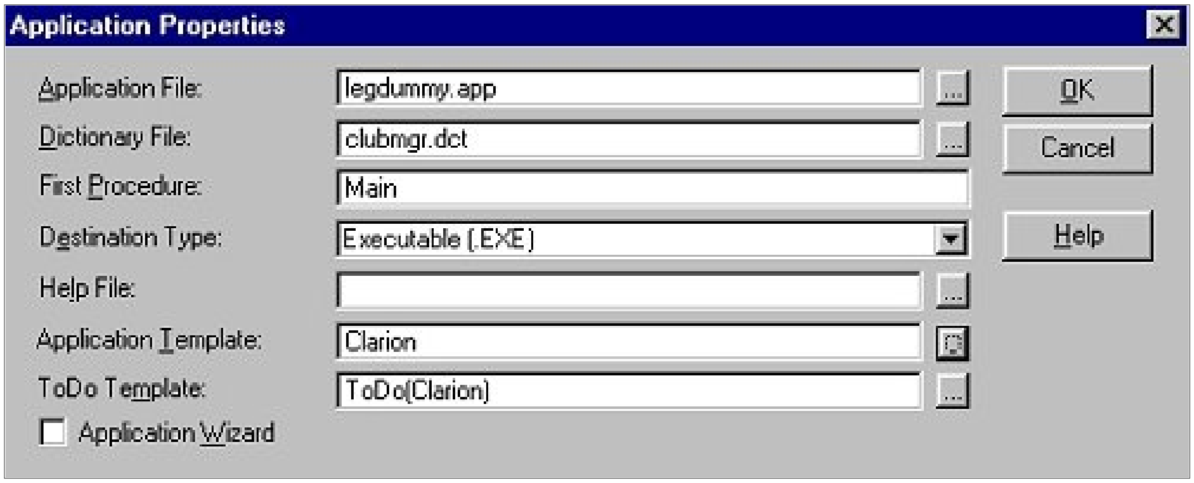
Ok let's start. If you don't know how the converter works, well, join the club! One thing I do know about the converter is that it converts a whole application from Legacy to ABC in one go. That's not your aim or mine, so there needs to be an interim step – the creation of a dummy application.

A Dummy Legacy Application

You can create a dummy Legacy application and transfer the procedures to be converted very simply following these steps:

1. Create a new Legacy (Clarion template chain) application specifying the ClubMgr dictionary but without using the Application Wizard, as shown in Figure 20.
2. When the empty application is created, from the Main Clarion menu choose File, Import From Application.
3. Select the Legacy application created way back in Part 1 from the file dialogue.
4. Select the four procedures mentioned above from the Select Items to Import dialogue.

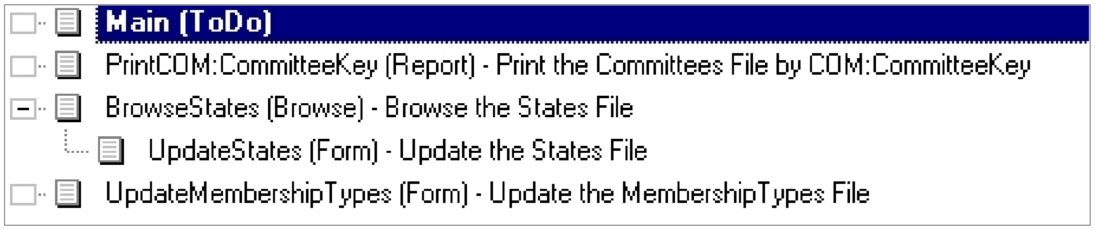
Figure 20. Creating Dummy Legacy Application.



Application File: legdummy.app
Dictionary File: clubmgr.dct
First Procedure: Main
Destination Type: Executable (.EXE)
Help File:
Application Template: Clarion
ToDo Template: ToDo(Clarion)
[ ] Application Wizard

Your dummy application tree should now look like that shown below in Figure 21.

Figure 21. Application Tree for Dummy Legacy Application.



Main (ToDo)
PrintCOM:CommitteeKey (Report) - Print the Committees File by COM:CommitteeKey
BrowseStates (Browse) - Browse the States File
UpdateStates (Form) - Update the States File
UpdateMembershipTypes (Form) - Update the MembershipTypes File

Convert or Bust!

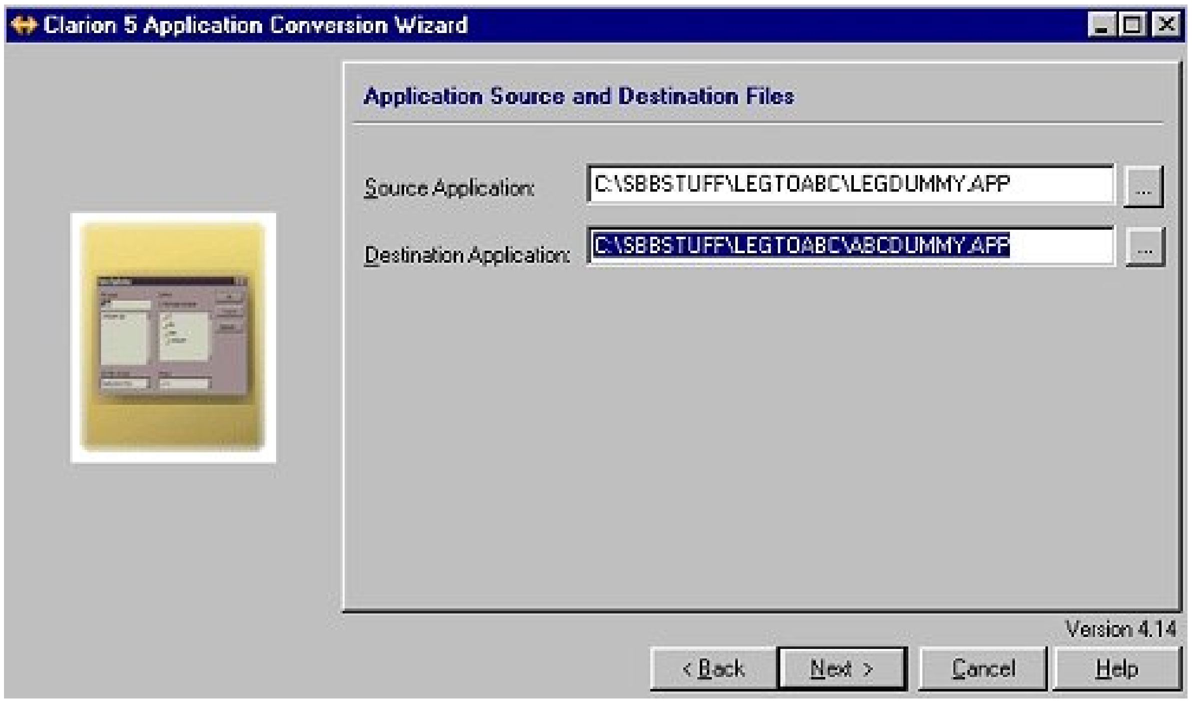
It’s now time to run the converter. I’ve found that the converter recreates most of my procedures quite well so I don’t need to tinker with it. I’m sure you’ll have a similar experience for most of your procedures; the others, well, I’ll just sympathise.

The converter program takes an entire Legacy application and creates an entire ABC application. For that reason, it's not possible for you to convert these procedures and place them directly into a larger app. Instead, you must go through another stage – a dummy ABC app.

To run the converter over the dummy Legacy application, do the following:

1. From the Main Clarion menu choose File, Convert Application.
2. Follow the Conversion Wizard through selecting defaults and choose a suitable name for your dummy ABC application, as shown in Figure 22.
3. I tend to hit the Apply button to the myriad of questions the converter asks me – you may need to be more scrupulous!

Figure 22. Application conversion wizard naming options.


Clarion 5 Application Conversion Wizard
Application Source and Destination Files
Version 4.14
< Back
Next >
Cancel
Help

Source Application: C:\SBBSTUFF\LEGTOABC\LEGDUMMY.APP
Destination Application: C:\SBBSTUFF\LEGTOABC\ABCDUMMY.APP

Finally you’ll end up with a converted ABC application which, on the surface, looks exactly the same as the dummy Legacy application. Under the covers it’s a very different story – these procedures are now ABC. That doesn’t mean they’ll work yet – they may have embedded code to alter, for example. But now's not the time to work on that problem; you must wait until they’re in your ABC Procedure DLL.

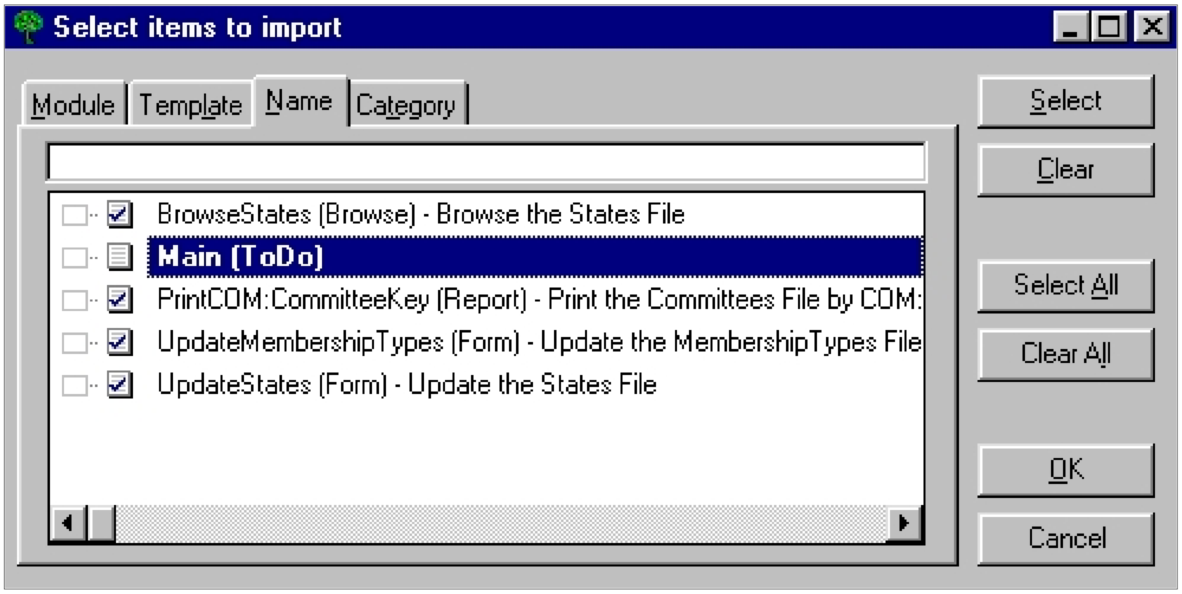
Transfer To ABC Procedure DLL

Remember the ABC Procedure DLL created during Part 2? That's where you first saw ABC procedures at work with your Legacy application. In that case, you created new ABC procedures. Today you're going to extend that DLL with the converted procedures from your Legacy app.

To do that, follow these steps:

1. Open the ABC Procedure DLL app created during Part 2.
2. From the main Clarion menu choose File, Import From Application.
3. Select the dummy ABC application from the next dialogue.
4. From the Select Items to Import dialogue, select the four converted procedures, as shown in Figure 23.

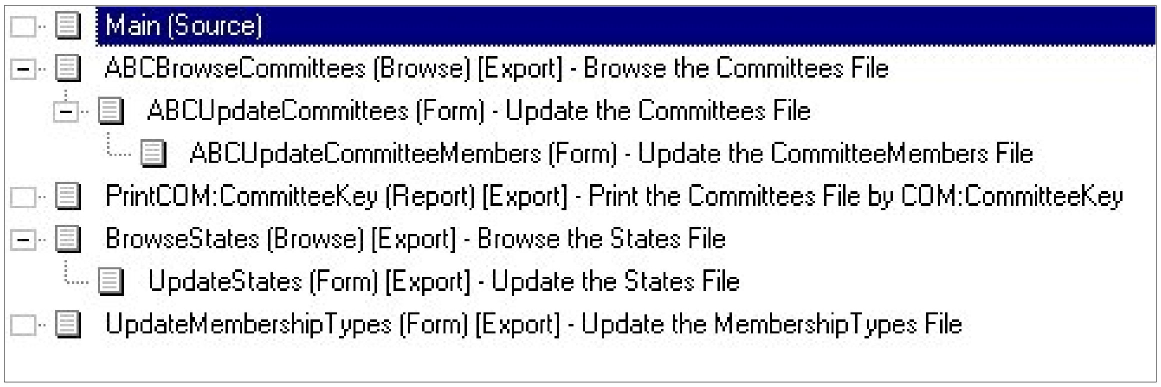
Figure 23. Selecting converted ABC procedures to import.



☑ BrowseStates (Browse) - Browse the States File
Main (ToDo)
☑ PrintCOM:CommitteeKey (Report) - Print the Committees File by COM:
☑ UpdateMembershipTypes (Form) - Update the MembershipTypes File
☑ UpdateStates (Form) - Update the States File

Your ABC Procedure App should now look like that shown in Figure 24.

Figure 24. ABC procedure app showing imported procedures.



```
Main (Source)
ABCBrowseCommittees (Browse) [Export] - Browse the Committees File
    ABCUpdateCommittees (Form) - Update the Committees File
        ABCUpdateCommitteeMembers (Form) - Update the CommitteeMembers File
PrintCOM:CommitteeKey (Report) [Export] - Print the Committees File by COM:CommitteeKey
BrowseStates (Browse) [Export] - Browse the States File
    UpdateStates (Form) [Export] - Update the States File
UpdateMembershipTypes (Form) [Export] - Update the MembershipTypes File
```

You'll notice that the `UpdateStates` form is showing as an exported procedure. This attribute need only be on if the procedure will be called from the Legacy app (or another DLL). In this case, you know that only the `BrowseStates` procedure will ever call it, so go into the Procedure Properties for `UpdateStates` and turn off the Export Procedure check box.

If you have embeds, now is the time to begin reviewing them and getting them ready for ABC. Some embedded code will no longer work, like calls to the `CheckOpen` function, the `ResetWindow` routine or the `ProcedureReturn` routine. There are far too many possibilities to mention here but you'll soon work it out. At least you now have the ability to do trial-and-error on a small, manageable scale.

When satisfied, Make this enlarged DLL.

The Final Link

It's time to call these newly converted procedures from the Legacy app. However, before that's done, you'll need to save a copy of the Legacy procedures you're replacing. There are a few ways to do that, but I find the least confusing method is to make a copy of the procedures under a new name within the Legacy app and leave the original procedure names as they were. If you rename instead of copy, you could end up preserving the link to the old procedure from, say, a main menu, and that's not what you want. Keeping the saved copy of the procedures within the Legacy app makes it very easy to revert back if required.

A side benefit of keeping the original procedure names is that if you have some embedded code which calls a procedure, it should be able to find it without further alterations.

To make a safety copy of the old Legacy procedures, follow these steps:

1. Open the Legacy app.
2. Highlight each procedure in turn and select Procedure, Copy from the main Clarion menu.
3. Give each copied procedure a meaningful name – good conventions would be to prefix the name with "old" or "leg".

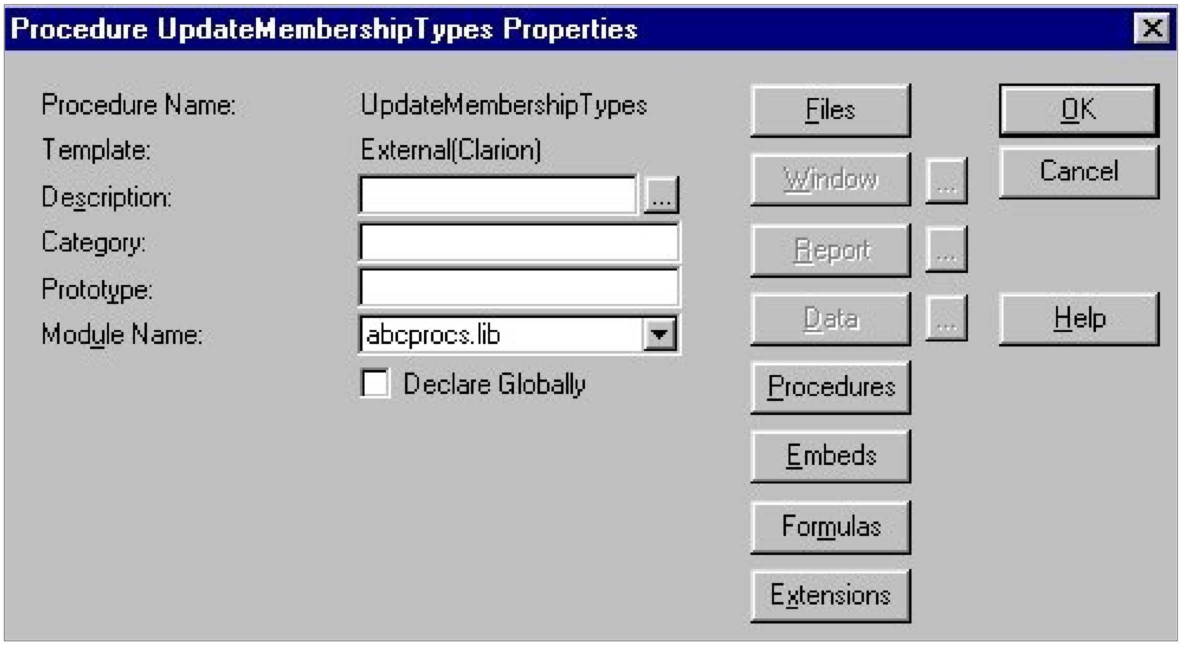
Having safely copied the original Legacy procedures, it's now time to delete the originals. Begin at the bottom of each calling tree and delete from there upwards. For example, given that you've converted the sub-tree containing the `BrowseStates` and `UpdateStates` procedures, the `UpdateStates` procedure is at the bottom of the tree so it gets deleted first followed by `BrowseStates`. Doing it this way will make it easier to ensure you don't end up with any duplicate procedures hanging around.

Having deleted the procedures you should now have three ToDo procedures in your tree, those being `BrowseStates`, `UpdateMembershipTypes` and `PrintCOM:CommitteeKey`. To re-invoke them using the ABC versions converted earlier, they must be linked to the ABC Procedure DLL library. You should remember back in Part 2 that you added the ABC Procedure DLL library to your Legacy app, so that's already available.

For each of these procedures in turn, perform the following steps:

1. Double-click on the procedure and choose External – External Procedure from the Select Procedure Type dialogue.
2. In the Procedure Properties window select the correct library from the Module Name drop-down list box, as shown in Figure 25.

Figure 25. Procedure Properties for External Procedure.


Procedure UpdateMembershipTypes Properties

Procedure Name: UpdateMembershipTypes

Template: External(Clarion)

Description:

Category:

Prototype:

Module Name: abcprocs.lib

☐ Declare Globally

You may now Make and Run the Legacy application. That's it, you're truly underway with your first conversion!

Note that when you're converting your own applications, the procedures you delete may not conveniently show up in the application tree as ToDo procedures. Not to worry, you just need to select Procedure, New from the main Clarion menu and continue from there.

Reversion

Let's say you weren't happy with the way one of these procedures is operating in ABC and you want to revert to the Legacy version. The easiest way to do that is to alter the "calling" procedure to call the saved Legacy copy of the procedure. If that isn't practical, follow these steps:

1. Go into your ABC Procedure DLL and turn off the Export Procedure check box on the Procedure Properties window of the procedure(s) in question.
2. Make the ABC Procedure DLL.
3. Open the Legacy application and delete the "External" version of the procedure(s). This may or may not leave a ToDo procedure in its place depending on the calling tree.
4. For each procedure you wanted to restore, copy the saved version of each procedure back to its original name – I mean Copy, not Rename.
5. Make the Legacy application.

Step one above shows up an interesting aspect of this hybrid method, or any multi-DLL application for that matter - you can't have two procedures with the same name in the same application if at least one of them is exported. To clarify, if you have a procedure Exported from a DLL, any other EXE or DLL that includes your library which has an identically named procedure will have a duplicate name conflict.

You may remember that one of the first steps when creating the DLLs during this tutorial was that you went into the Main procedure, made it a Source procedure and turned off the Export Procedure check box. It was to avoid exactly the type of conflict mentioned above.

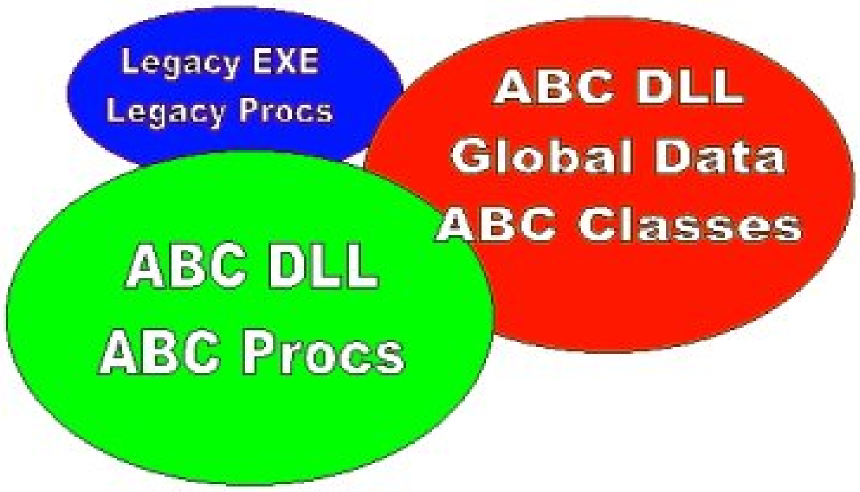
If you do need to revert remember that you must not break the Golden Rule principles.

Ongoing Conversion

In case it's not obvious to you, the conversion you've been carrying out so far can be done in a continuous loop until you have no more procedures left to convert. That is, you re-create the Dummy Legacy application over again, import some more Legacy procedures to it, convert to Dummy ABC, import to the ABC Procedure DLL, save the original Legacy procedures and make the link to the ABC DLL. Then do it again, and again, and again.

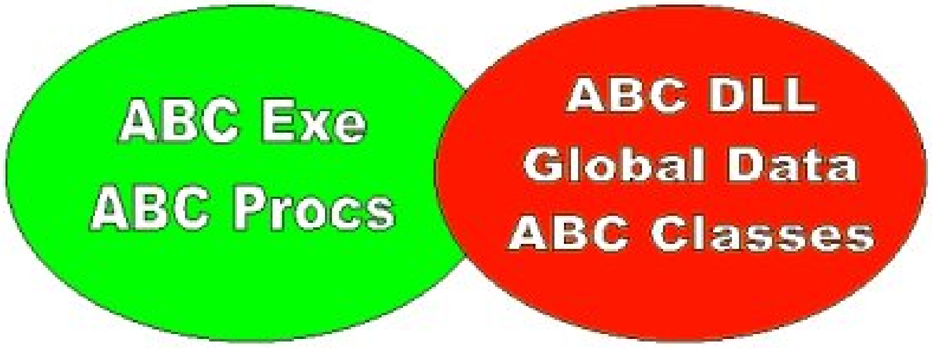
Your application will soon start looking like Figure 26, as the ABC portion of your application increases and the Legacy portion decreases. If procedures don't convert to your satisfaction or you're worried about them, rewrite them from scratch within the ABC Procedure DLL rather than converting them.

Figure 26. Advanced stage of hybrid conversion.



Finally, you can entirely dispense with the Legacy part of the application and enjoy pure ABC as shown in Figure 27.

Figure 27. Final Application.



Final Hints

One important item that I haven't yet covered is your new-found ability to introduce inconsistency into your app using the hybrid method. A great example of this is in the hybrid application you've just developed, so start it up.

In the Reports menu choose Print By Committee, which starts the ABC report you converted earlier. If your Clarion templates are at default values, the Print Preview window will come up at design (default) size. Now choose any other report (which is obviously Legacy) from the menu and the Print Preview window will come up maximized.

Why? Well the ABC side of your app is using the ABC `PrintPreview` class while the Legacy side of your app still uses the old `ReportPreview` function in the Standard Functions template. In other words, they're two very different functions and, in this case, their default behavior is different.

This is not an isolated case – it's quite possible that any template-based function could differ between the two template chains. However, you're only going to be able to determine that by trial and error. In most cases, as in the example above, they will be small and fixable if they cause problems.

Another case where this could occur is in the use of third-party templates where the ABC version could well be much advanced over its Legacy equivalent. That may have other ramifications too, such as your code not converting properly. I suggest you seek advice from the supplier of the templates if you encounter any problems.

Conclusion

This tutorial has been drawn out deliberately to explain every aspect of conversion quite explicitly. If you're reading this and thinking "I must've missed something, it seems too straightforward" then you probably haven't missed anything.

If you're confused then chances are you've never dealt with a multi-DLL application before. All I can say to you is that attention to the finer points of the tutorial is vital to make it work; the method shown here will work for your application if you follow the instructions to a tee (the only unknown being the converter program).

I should also point out that it is possible you may confront problems I have not encountered in my testing. If you do, I implore you to stick with it and get over the hurdles – remember, this is Clarion and you have a formidable arsenal of possibilities in your hands; use them. Always test carefully and move forward, not backward. Work in small sections and isolate the problem areas. You will succeed.

I sincerely hope this article has helped solve your Clarion upgrade problems. Goodbye Legacy, hello ABC!

Simon Brewer is Software Development Manager for First Ecom, an Internet development company using Clarion. Prior to that he spent 17 years at Email Major Appliances, major Clarion users and Australia's largest manufacturer of whitegoods. In his spare time he is also the President of the South Australian Clarion User Group and a co-organiser of the ConVic conferences.

Code Documentation: The Achilles´ Heel Of Clarion

by Steffen Rasmussen

Documentation is essential for maintaining an application. Unfortunately there is no way to automatically document code in Clarion, and manual documentation is laborious and prone to errors. Programmers are compelled to keep a strict discipline in order to maintain consistency. Because of the involved labour in many instances there isn't any code documentation at all and if there is, it's insufficient. As a result a lot of time is consumed in code "interpretation" and in extreme cases, reprogramming the whole application is a necessity.

So where does the programmer document the code? Well there aren't separate template prompts for embedded source comments, so it will have to be in the source itself. Apart from this the programmer has to understand how to use the tools in the programming environment in order to maintain a manageable code base.

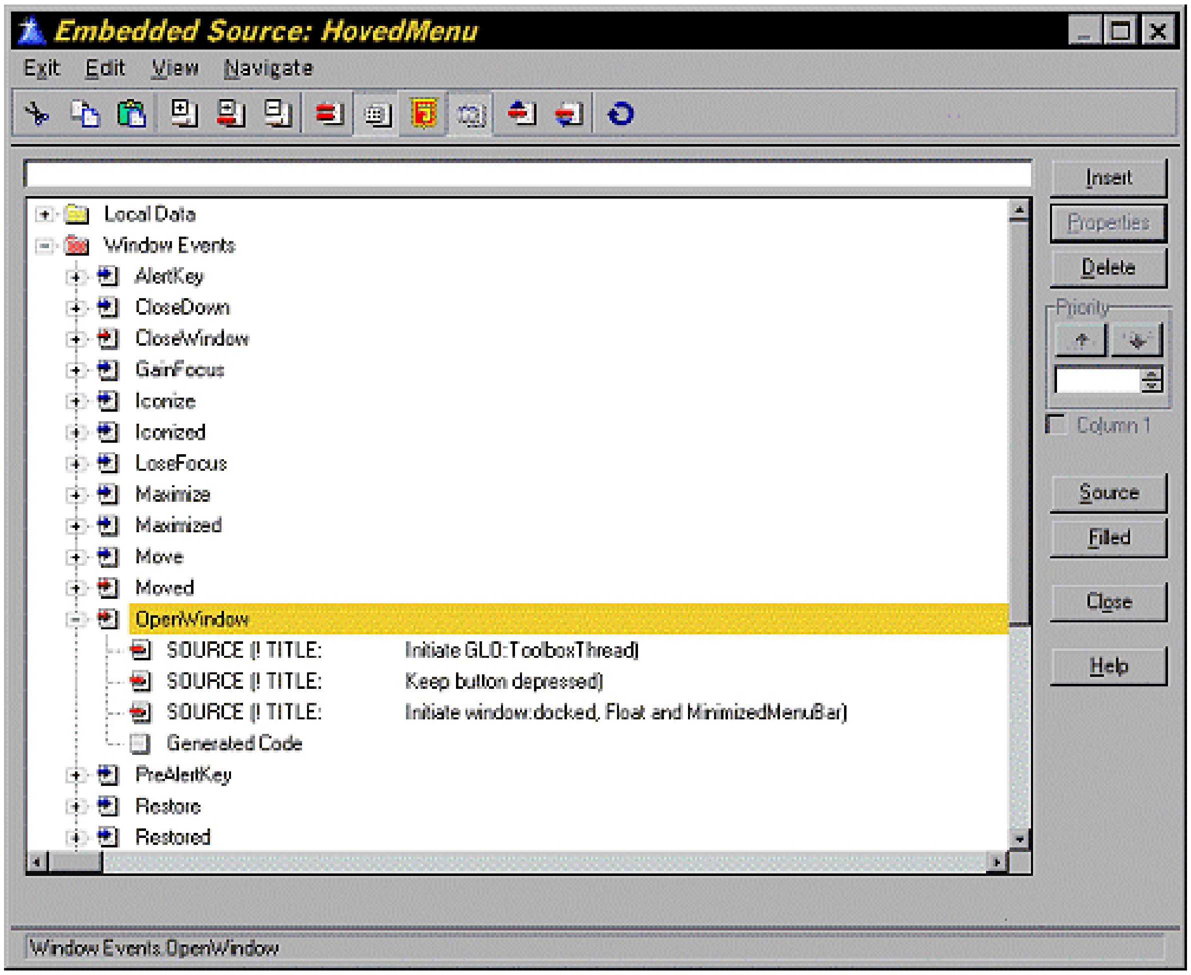
The whole programming environment consists of individual sections of source code interacting with each other. When the programmer implements source embed into this structure he or she has to keep in mind that it has to be an individual unit. In other words even though there is a lot of different code in exactly the same embed point with the same priority, each functional group should be kept in its own source embed. In this way one might have 10 different source modules in the same place. This can be used to enhance the code readability in the embed tree.

Figure 1. The menu's embedded source.

Developer PLUS

Your source for development tools and add-ons.

Your outlet for application sales.



```
Local Data
Window Events
    AlertKey
    CloseDown
    CloseWindow
    GainFocus
    Iconize
    Iconized
    LoseFocus
    Maximize
    Maximized
    Move
    Moved
    OpenWindow
        SOURCE (! TITLE:        Initiate GLO:ToolboxThread)
        SOURCE (! TITLE:        Keep button depressed)
        SOURCE (! TITLE:        Initiate window:docked, Float and MinimizedMenuBar)
        Generated Code
    PreAlertKey
    Restore
    Restored
```

Window Events.OpenWindow

One way to document source code is to start with a predefined header in the source that informs the programmer what this code contains. In this way the programmer can quickly form an idea about the code with out actually reading it.

So what does this predefined header contain?

Well as a minimum it should have the following:

TITLE: This is the name of the source block. The name should be as descriptive as possible so the programmer quickly can determine if this is the source he or she is looking for. Notice that by having the title in the first row (see screen dump above) it's much easier for the programmer to get an overall impression of the source blocks.

PURPOSE: Here the programmer uses a couple of lines to describe the purpose or objective of the module. Pseudocode can be used for this task.

INPUT: What kind of input does this source get.

OUTPUT: What kind of output does this source produce.

VARIABLES: Which variables are used by this module. The variable name is written as specified in the local data section of the procedure. Global variables are likewise written as in the global data section of the program. Personally I have chosen to make it easy to distinguish between the two variable types by having the global variables to be preceded with the extension GLO:, e.g. GLO:ToolBoxThread.

EMBEDS: As the application evolves so does the embedded code which is scattered all over the program. Therefore it is essential that each source block keeps track of the other procedures and modules it interacts with, e.g. by storing the name and locations of those other procedures and modules.

PROCEDURE SETINGS: Are there any local properties in this procedure which have to be taken into account when creating this code? State the property and the setting.

AUTHOR(S): Where does this module originate from, e.g. name of the programmer, book or other resource.

REMARKS: Contains any information that can clarify the code for the reader, including code history if appropriate. Instead of using remarks, the module header could if desired be more detailed and contain, for example, the module number, version number, date, date modified, who modified it, what was modified etc. Keep in mind, however, that although these extensions to the module header could be an advantage they can also have a downside. In essence keeping documentation to a minimum reduces the possibility of inconsistency in the documentation when updated.

Apart from the module header it is also very important to document each line of code as appropriate.

Let me introduce you to a small example on how documentation can be done:

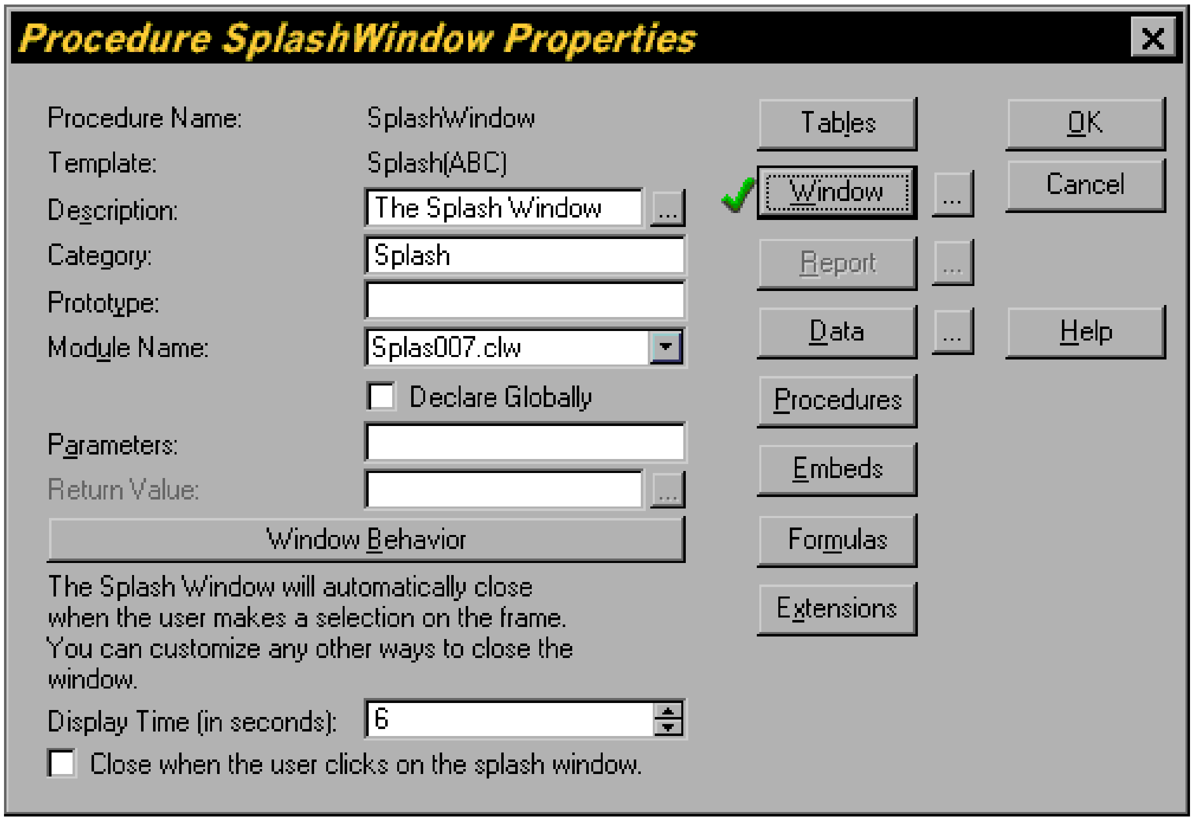
Creating A Splash Screen, With Documentation

In Clarion the programmer has the option to automatically create a splash screen. When the application starts the splash screen appears right after the frame. There are a lot of other applications out there that show the splash screen while the application is being loaded, and when loading is finished the splash screen disappears. So how do I get the same functionality?

Start off with what is known. The Main properties contains a procedure splash screen extension where it's possible to define the name of the splash screen. Call it SplashWindow.

In the procedure SplashWindow properties I will let the Display Time (in seconds) be 8. Since I don't want the user to close the splash window by clicking on it, I uncheck the check box. Now I have an application with a splash screen that appears after the program has started.

Figure 2. SplashWindow properties.


Procedure SplashWindow Properties

Procedure Name:	SplashWindow
Template:	Splash(ABC)
Description:	The Splash Window
Category:	Splash
Prototype:	
Module Name:	Splas007.clw
	☐ Declare Globally
Parameters:	
Return Value:	

The Splash Window will automatically close when the user makes a selection on the frame. You can customize any other ways to close the window.

Display Time (in seconds): 6

☐ Close when the user clicks on the splash window.

I can’t change the way different parts of the program are called during start up, so instead I will hide the application frame until the splash screen is finished.

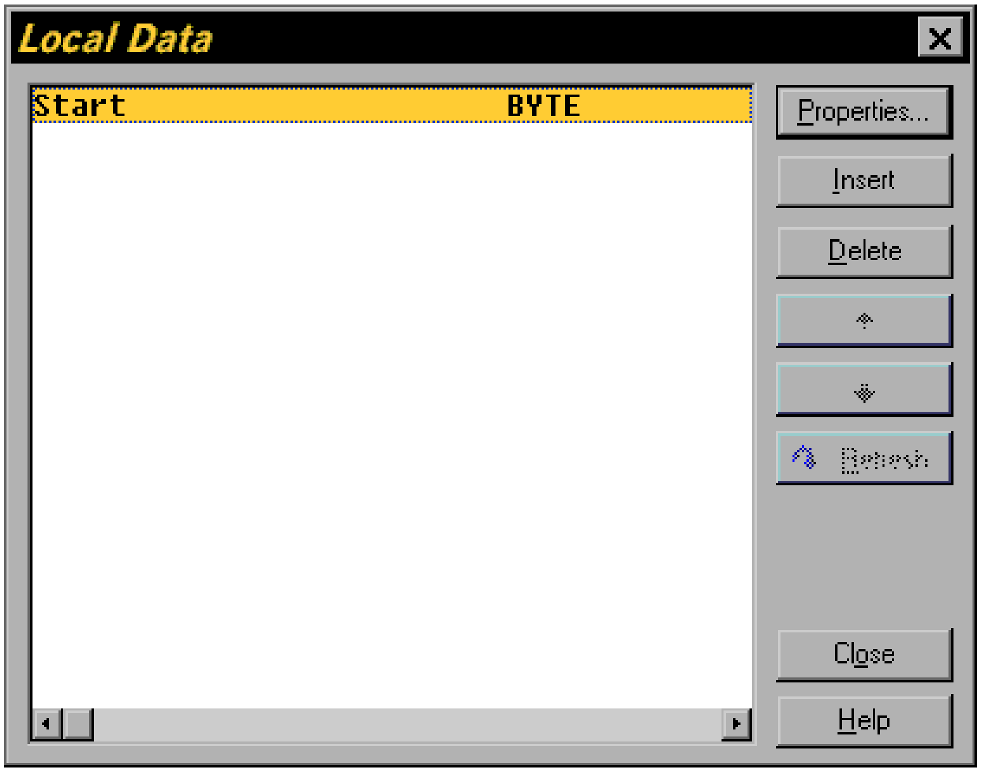
To hide the window I choose Embeds in the frame procedure. Here I will have to find the point where the window is opened so it can be hidden. In Local Objects. ThisWindow (WindowManger).Init PROCEDURE().CODE.Open the window. I place the source code

```
TARGET{PROP:Timer} = 100
TARGET{PROP:Hide} = 1
```

Now the window is hidden and timer activated. TARGET is a built-in variable in Clarion that normally refers to the window that currently has focus. TIMER can be omitted if the extension template "Display the date and/or the time in current window" is used.

The next step is to unhide the window. There should be a smooth transition between closing the splash window and unhiding the window. One way to do this is to use the timer. First of all I’m going to define a local variable Start in the data section of the Frame. This variable will keep track of the time.

Figure 3. Creating the Start local variable.



Start BYTE

In the Embeds for the frame select Local Objects.ThisWindow (WindowManager).TakeWindowEvent PROCEDURE()CODE and after Top of CYCLE/break support place the following code in the SOURCE embed:

```
OF EVENT:Timer
  CASE Start
  OF 1
  OF 2
  OF 3
  OF 4
  OF 5
      TARGET{PROP:Hide} = 0 !UnHide Window Frame
  OF 6
  OF 7
  OF 8
  OF 9
  END
  IF Start<10   !Start is only used the first 10 seconds
      Start+=1  !Increment Start with 1 second
  END
```

That's it…nearly. Although I have documented some of the code lines I'm still missing the code description in the header as described earlier. One could argument that because there isn't so much code it's superfluous. But even though this is just the preliminary phase of the program it is essential to start documenting at once.

I may as well start with the first embedded source: Local Objects.Abc Objects. Window manger. Init PROCEDURE().CODE.Open the window.

TITLE:

Hide window

PURPOSE:

Hide window while loading program and show splash screen.

INPUT: -

OUTPUT:

TARGET{PROP:Hide} = 1

VARIABLES:

- **EMBEDS:** Local Objects.Abc Objects.Window Manager.Init PROCEDURE. | CODE. TITLE: UnHide window.

PROCEDURE SETINGS:

Splash procedure: SplashWindow

AUTHOR(S):

Steffen S. Rasmussen

REMARKS:

TIMER can be omitted if the extension template' Display the date | and/or the time in current window' is used. In the embedded source: Local Objects.Abc Objects.Window Manager.Init PROCEDURE.CODE. Top of CYCLE/break

TITLE:

UnHide window

PURPOSE:

To trigger the event when the main window is unhidden.

INPUT: -

OUTPUT:

TARGET{PROP:Hide} = 0

VARIABLES:

Start

EMBEDS:

Local Objects.Abc Objects. Window manger.Init | PROCEDURE().CODE.TITLE: Hide window

PROCEDURE SETINGS:

Splash procedure: SplashWindow

AUTHOR(S):

Steffen S. Rasmussen

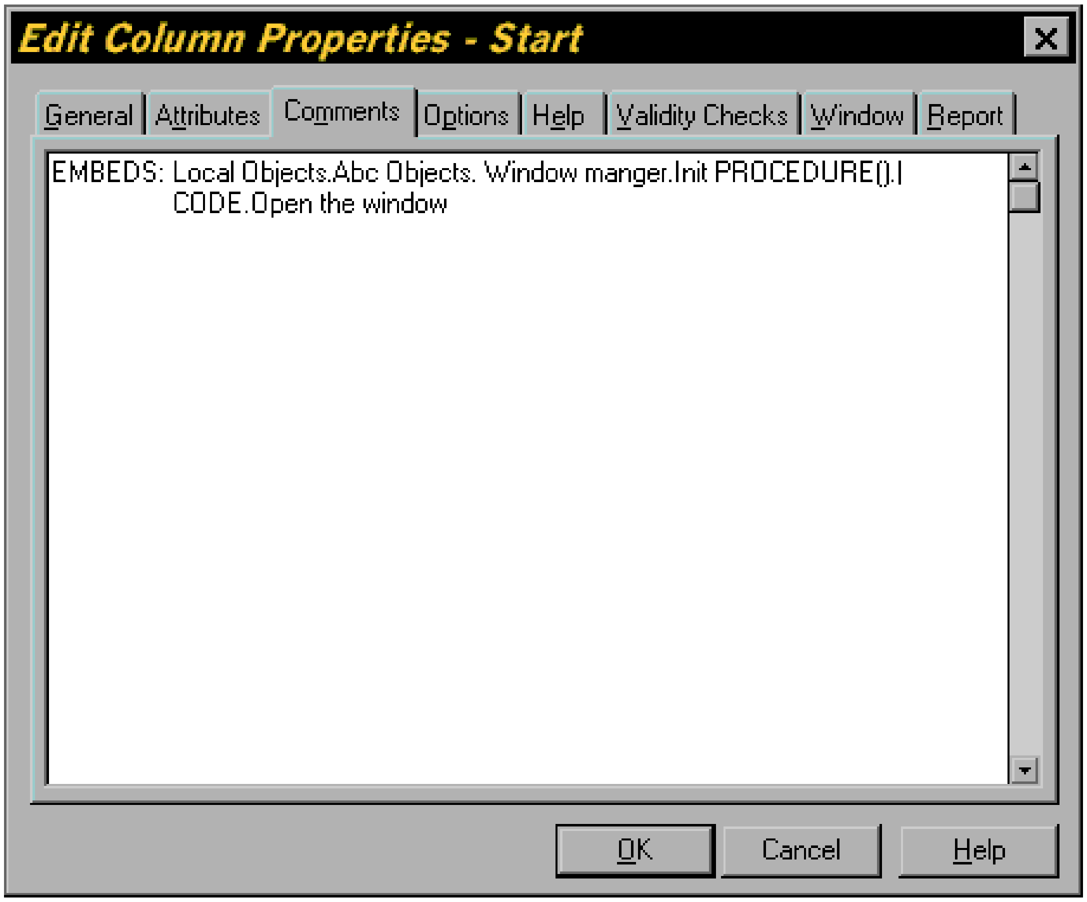
REMARKS:

For readability there is used a case structure that contains a trigger point for each second. In each trigger point the programmer can chose when and what is to be initiated. In this case the window is unhidden after 5 seconds. After 10 seconds the trigger is disabled.

After documenting all the source the last place to document this code is in the Local

Variable Start:

Figure 4. Adding comments to the Start variable.



```
EMBEDS: Local Objects.Abc Objects. Window manger.Init PROCEDURE().|
           CODE.Open the window
```

That’s it.

Although it is a tedious job, documenting code is a necessity that will pay back in the long run. Hopefully there will come a time when Clarion can assist the programmer in code documentation and maintenance to a much larger extent than it does today. Until then documentation will always be the Achilles´ heel of Clarion.

Download the example app

Steffen S. Rasmussen has graduated in Computer Science from Copenhagen Business College. Since then he has worked as a programmer, system technician and network administrator, and is currently IT manager. Clarion is a quite a new language to Steffen since his only been working with it since January 2000. But what better way to learn it than by trying to teach others! Steffen has also set up a web site to collect as many examples of different user interfaces as possible to inspire Clarion developers.

Reborn Free
CLARION online
published by CoveComm Inc.
Clarion MAGAZINE

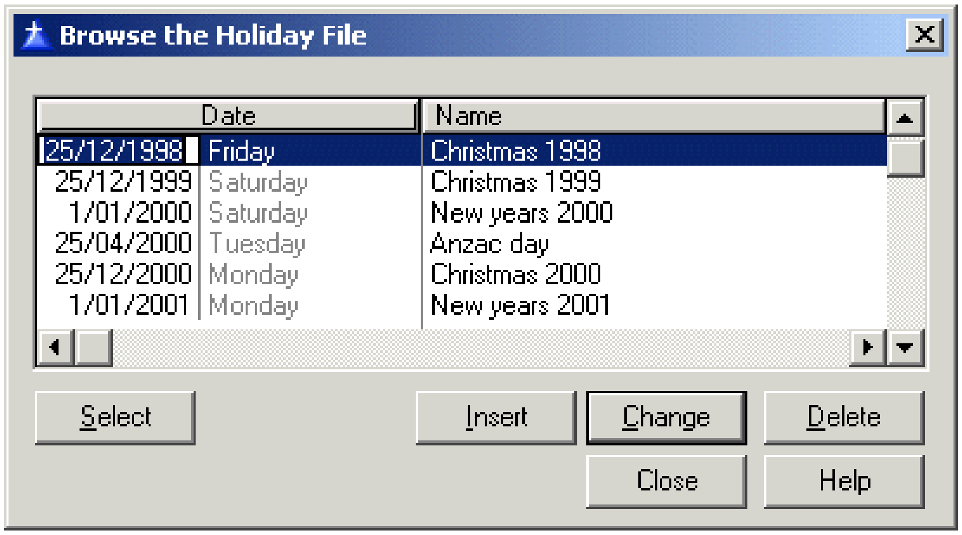
Displaying Related Fields In ABC Edit-In-Place

By Alan Telford

Edit-In-Place (EIP) is really useful but requires some effort for all but the most basic of requirements. Consider the case where the user has a browse with a date field.

Do you know which day of the week 3 May 2000 was? Neither does the user. So the user likes to have the day name displayed automatically next to the date field. The date field is an editable field from the database, whereas the day name is a local data field for display only. How can you setup EIP so that the day name is always displayed correctly?

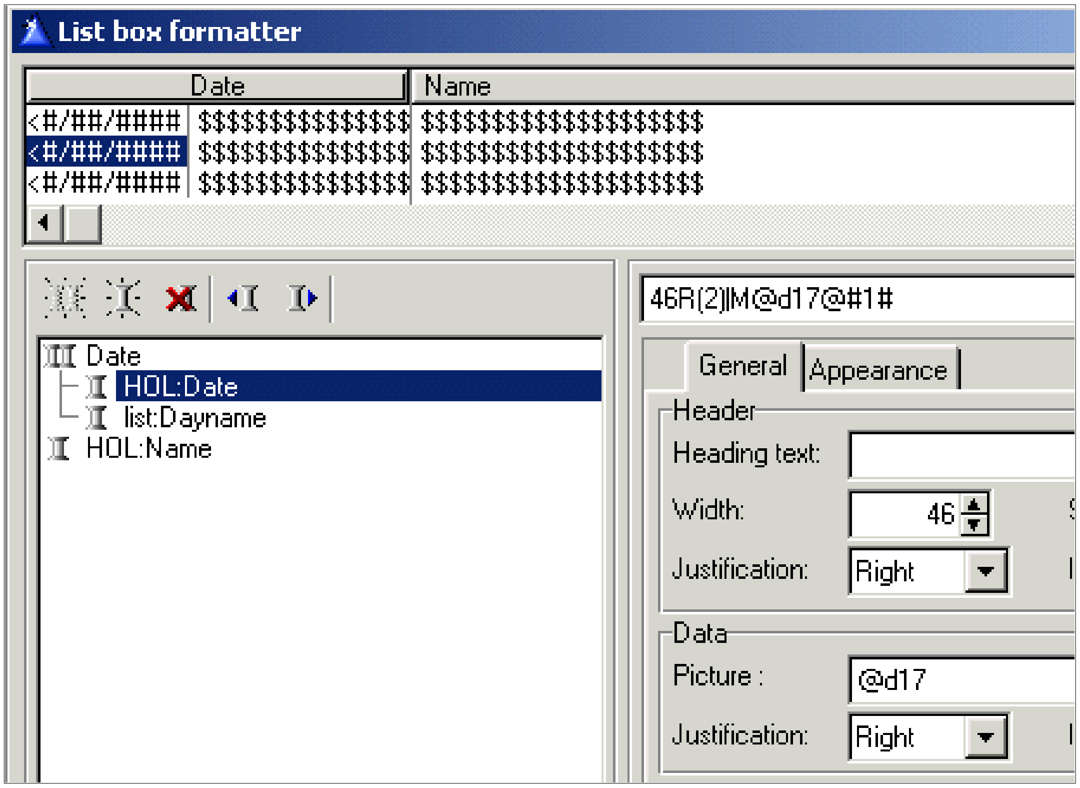
Figure 1. Browse with date and day name.



Browse the Holiday File

Date		Name
25/12/1998	Friday	Christmas 1998
25/12/1999	Saturday	Christmas 1999
1/01/2000	Saturday	New years 2000
25/04/2000	Tuesday	Anzac day
25/12/2000	Monday	Christmas 2000
1/01/2001	Monday	New years 2001

Figure 2. List box formatter view of browse.

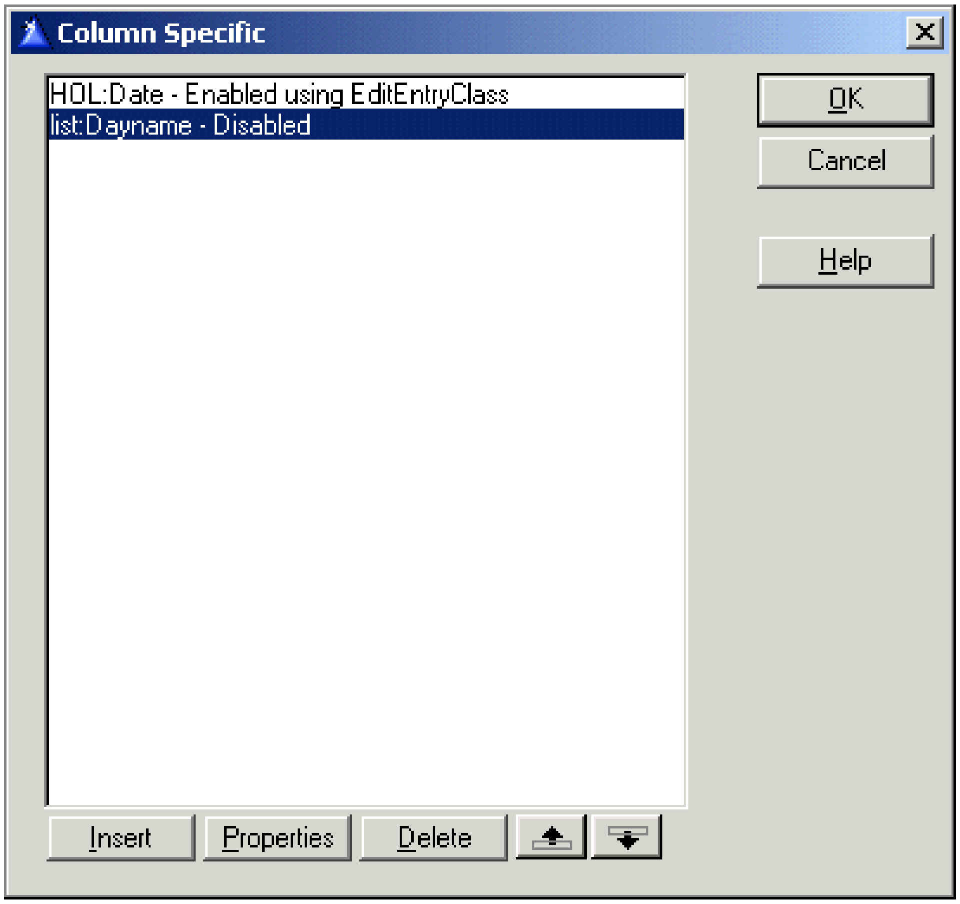


46R(2)|M@d17@#1#

Width: 46
Justification: Right
Picture : @d17
Justification: Right

On your browse, click on the Change button, select Configure Edit in Place, and then select Column Specific.

Figure 3. Column specific settings.



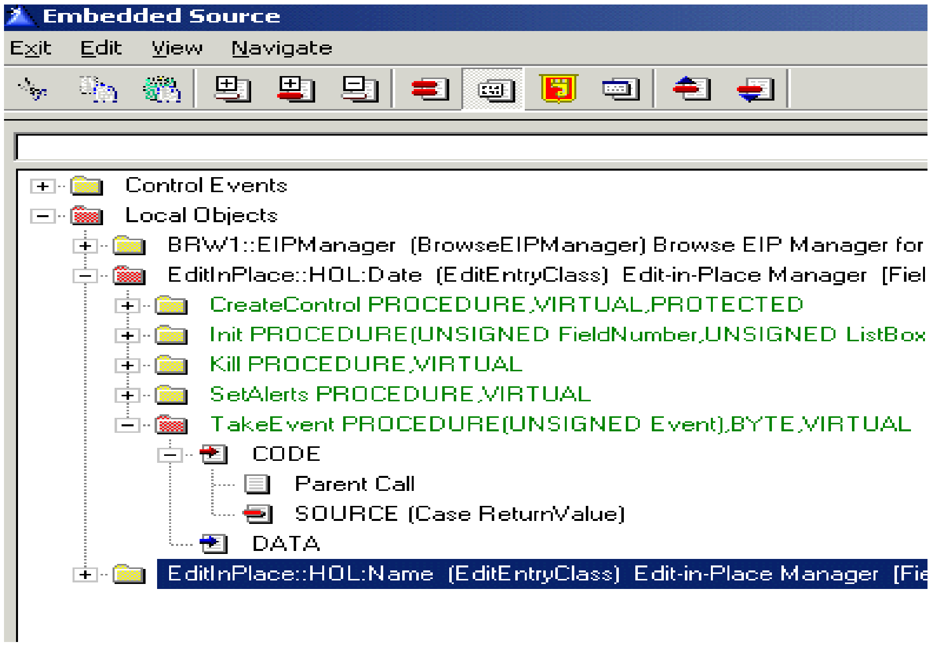
HOL:Date - Enabled using EditEntryClass
list:Dayname - Disabled

In my example, `HOL:Date` is the database field for the date. This should be enabled using the standard `EditEntryClass`.

`List:Dayname` is the local data field used to display the day name in the browse and during EIP.

Next you have to embed the actual code. Go to the embed list. Under local objects, find the `EditInPlace::HOL:Date` object, and then embed the following code after the parent call of the `TakeEvent` method.

Figure 4. Embedding code to do the lookup.



```
Control Events
Local Objects
  BRW1::EIPManager (BrowseEIPManager) Browse EIP Manager for
  EditInPlace::HOL:Date (EditEntryClass) Edit-in-Place Manager [Fiel
    CreateControl PROCEDURE,VIRTUAL,PROTECTED
    Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox
    Kill PROCEDURE,VIRTUAL
    SetAlerts PROCEDURE,VIRTUAL
    TakeEvent PROCEDURE(UNSIGNED Event),BYTE,VIRTUAL
      CODE
        Parent Call
        SOURCE (Case ReturnValue)
      DATA
  EditInPlace::HOL:Name (EditEntryClass) Edit-in-Place Manager [Fie
```

The embed code is as follows:

```
Case ReturnValue
OF EditAction:None
OROF EditAction:Cancel
ELSE
  ! equivalent of EVENT:Accepted
  Update(SELF.Feq)
  Brw1.Q.list:Dayname = NameOfDay( Brw1.Q.HOL:Date )
  PUT(Brw1.Q)
  display()
END
```

Note that the real code is wrapped about with a `Case ReturnValue` statement. This changes the embed point into the equivalent of `Event:Accepted` for a field on a Form template. But notice the subtle differences. Instead of:

```
list:Dayname = NameOfDay(HOL:Date )
display()
```

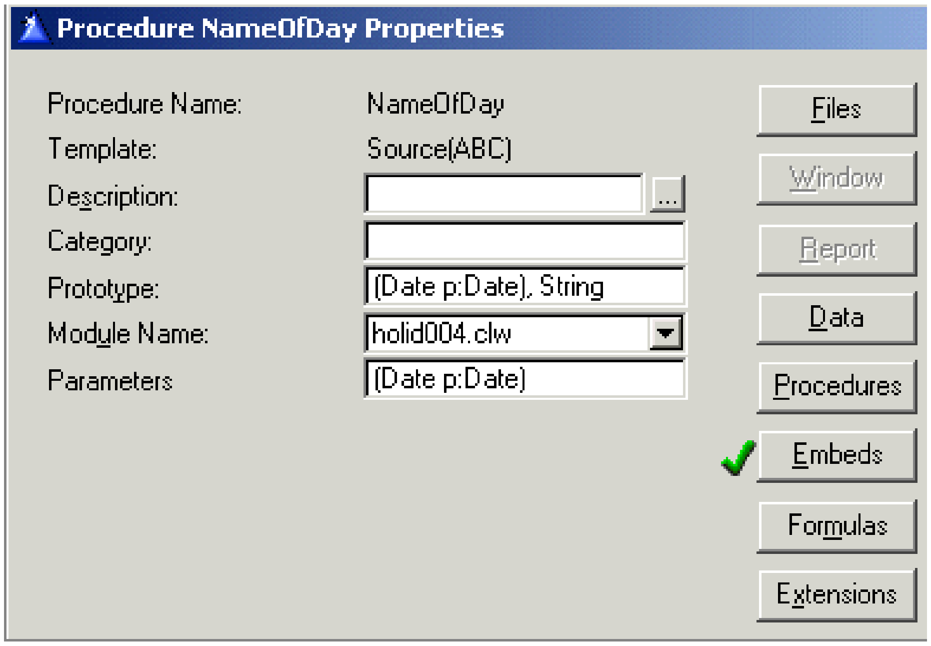
you actually have:

```
Brw1.Q.list:Dayname = NameOfDay( Brw1.Q.HOL:Date )
PUT(Brw1.Q)
display()
```

In EIP you are updating the browse queue and not the actual fields, so you must prefix both `HOL:Date` and `list:Dayname` with `Brw1.Q.` And don't forget the `PUT(Brw1.Q)` which ensures the listbox displays accurately upon exiting EIP.

Did you notice I've cheated slightly? I've used the `NameOfDay()` function. To create this, Insert a new procedure from the main application tree and call it `NameOfDay.` Enter the prototype and parameters as in Figure 5.

Figure 5. Name Of Day function.



Procedure NameOfDay Properties

Procedure Name: NameOfDay
Template: Source(ABC)
Description:
Category:
Prototype: (Date p:Date), String
Module Name: holid004.clw
Parameters: (Date p:Date)

Under the Processed Code embed point enter:

```
Return choose(p:Date%7+1, 'Sunday','Monday','Tuesday',|
'Wednesday','Thursday','Friday','Saturday')
```

So now your user can see the day name as soon as they have entered a date. One more happy user!

Alan Telford has been programming in Clarion since 1994. He is the Chief Software Developer at Maxtel Software Ltd, a New Zealand software company specializing in writing back office computer solutions for McDonald's Family Restaurants and other similar markets.

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

Legacy to ABC: There is Another Way!

by Simon Brewer

Part 2

In part one of this article I discussed the reasons why many of us still have Legacy applications, and offered some hope for their future in an ABC world. I showed how to begin converting a Legacy application to a hybrid Legacy/ABC application. I'm hoping I made good headway in dispelling the old myth that ABC conversions are difficult! In this article I'll explain how to build some pure ABC into the hybrid application and take the next step towards conversion.

Beware: Open Files!

Before starting on the code, I'll digress and talk about file "open status" handling, which is (hopefully) the only stumbling block you'll encounter.

In Legacy, each open file is tracked using a global counter variable named *filename*::used. These counters increment every time a file is required and decrement whenever it is finished with. If it reaches zero, the corresponding file is closed.

ABC uses different variables (properties) nested within the `FileManager` class, yet the hybrid application you created in Part One didn't complain that it couldn't find the older style global variables. Why? Because the simple act you did of setting the global **When Done With Files** option to **Keep File Open** conveniently de-referenced those variables – they're now ignored (for lack of a better word!) by the compiler.

Unfortunately, there is no equivalent to the **When Done With Files** setting in ABC. Don't go looking for it because it's not there. In fact, the logic for closing files in ABC is buried quite deep within the `FileManager` class and can not be over-ridden. That's quite a problem because it could allow an ABC procedure to close a file that's still in use by an active Legacy procedure.

The fix requires a one-line change to the `FileManager` class source file. I know, I know, no one should ever alter the templates or classes shipped with Clarion. However, the benefits far outweigh the potential risk in this particular case.

To make the change, open the abfile.clw file in your \clarion5\libsrc (or c55\libsrc) directory and search for the statement `CLOSE(SELF.File)` of which there will be only one occurrence. Simply comment the statement by placing an exclamation mark in front of it and save the file. You'll need to re-make the ABC Global Data DLL from Part One to bring this change in. That wasn't too hard now, was it?

To recap, the change made above will mean that any file opened in every ABC application you write will remain open until the application is closed or you execute an explicit CLOSE(*file*) source statement. Thankfully, the constraints once imposed by networks and DOS are mostly gone, so generally speaking this change is not going to cause any problems. However, it's important you understand the implications.

The Golden Rule

Before I continue with this part of the tutorial, I'll lay down a single conversion rule that I suggest you follow. That rule is to start conversions or make ABC additions to your hybrid Legacy/ABC app from the bottom of the Application Tree upwards. In other words, Legacy procedures may call ABC procedures; ABC procedures should only call other ABC procedures.

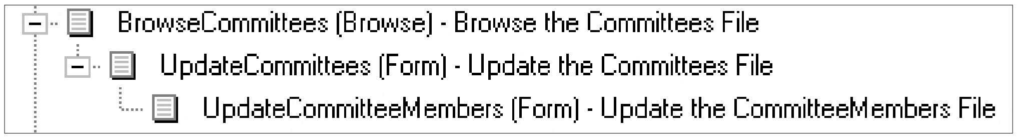
There are three good reasons for this rule:

1. It is much easier to understand when done this way.
2. It helps limit the number of entry points to the DLL ("Exported" procedures).
3. A DLL cannot call a procedure located in an EXE.

The latter point is fundamental to Clarion apps, whether Legacy or ABC. The hybrid method I'm demonstrating is based upon a Legacy EXE calling procedures in an ABC DLL, so this constraint applies. The other points are generally helpful reasons for adopting this rule.

To illustrate, please refer to the Application Tree snippet shown in Figure 1. It's an extract from the Legacy app you created in part one. Applying the rule to this extract, the procedures `UpdateCommitteeMembers`, `UpdateCommittees` and `BrowseCommittees` must be converted to ABC in that order. They can, of course, be converted all at once.

Figure 1. Extract from Legacy Application Tree.



BrowseCommittees (Browse) - Browse the Committees File
UpdateCommittees (Form) - Update the Committees File
UpdateCommitteeMembers (Form) - Update the CommitteeMembers File

If any of these procedures had embedded code calling another Legacy procedure, one not shown in the Application Tree, that procedure would also have to be converted. For example, in my apps I often use a small procedure named `Unxer` to capture unexpected error conditions after hand-coded file operations. Because I call it so often, I tend not to update the Application Tree for it. If I had made any call to that procedure within the `UpdateCommittees` form, I'd need to convert the `Unxer` procedure to ABC before, or with, the form.

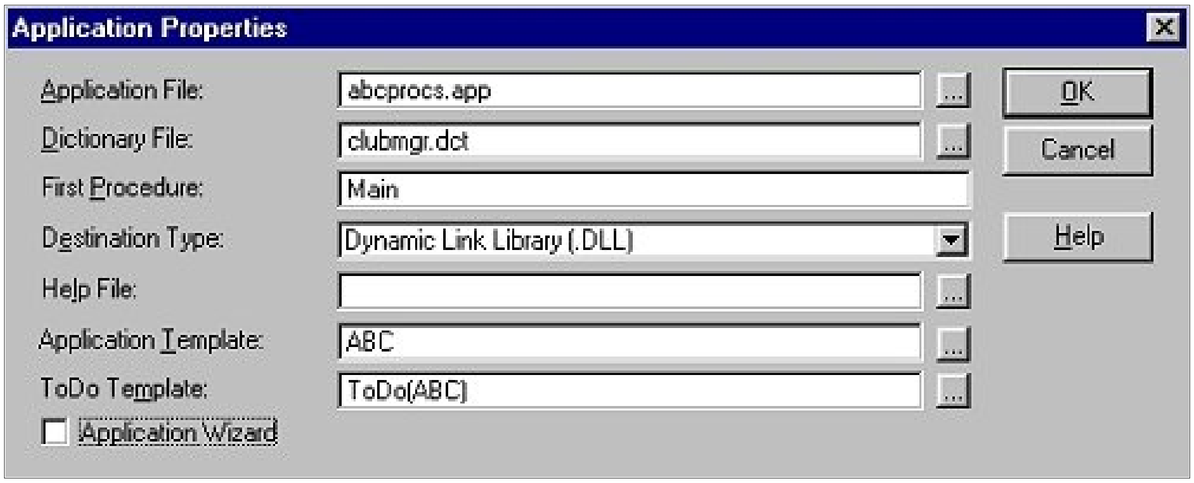
The ABC Procedure DLL

To begin procedure conversion, you'll need to extend the hybrid Legacy/ABC app by adding another DLL. Although you could technically begin adding procedures to the ABC Global Data DLL created in part one, it is more beneficial to leave that alone and create a new one. Therefore, you'll need to create another empty ABC DLL in a similar manner to which you created the ABC Global Data DLL last time.

Follow these steps:

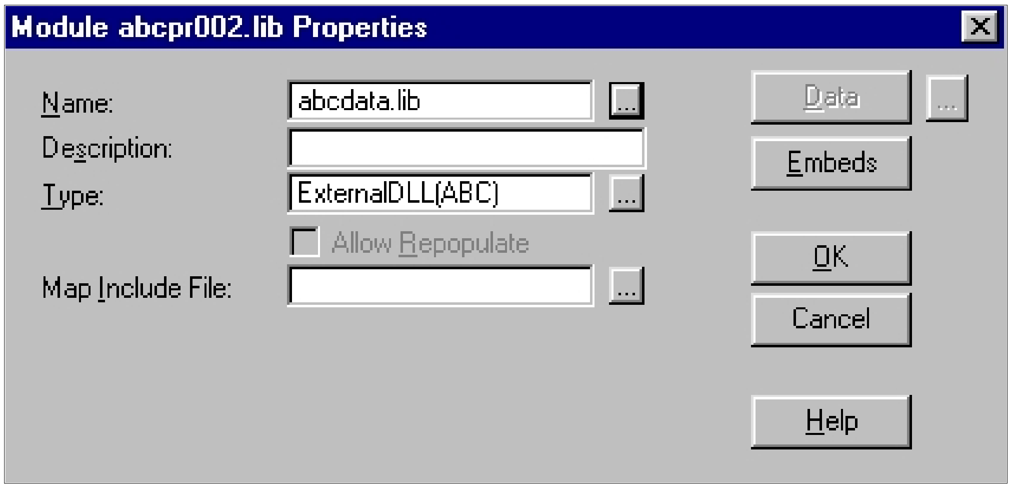
1. Create a new ABC application as shown in Figure 2. Specify the ClubMgr dictionary, set the **Destination Type** to **Dynamic Link Library (DLL)** and do not use the Application Wizard.
2. When the application is created, make the Main procedure an empty *Source* procedure and turn off the Export Procedure check box on its **Procedure Properties** window.
3. Under the **Global** section, in the **General** tab, check the **Generate Template Global Data as External** check box.
4. In the **File Control** tab under **File Attributes, External** set to **All External** and check the **All Files Declared in Another APP** check box.
5. From the **Application** section of the main Clarion menu, select **Insert Module**, choose **External DLL** and insert the Global Data DLL library name for the DLL created in part 1 (Figure 3).

Figure 2. Creating New ABC Procedure DLL.



Application File: abcprocs.app
Dictionary File: clubmgr.dct
First Procedure: Main
Destination Type: Dynamic Link Library (.DLL)
Help File:
Application Template: ABC
ToDo Template: ToDo(ABC)
☐ Application Wizard

Figure 3. Inserting ABC Global Data DLL Library.



Name: abcdata.lib
Description:
Type: ExternalDLL(ABC)
☐ Allow Repopulate
Map Include File:

You may now Make this DLL. If you're performing a conversion of a *real* Legacy app with a number of global data declarations, and you need access to those declarations in the ABC equivalent procedures that will reside in this new DLL, now is the time to copy them over. You should use the ellipsis (…) button to the right of the Global Data button to perform this task. If you copy them from the Legacy app that already includes this ABC Global Data DLL, each declaration will already have the External – DLL *Storage Class* on it as described in part one.

The First ABC Procedure

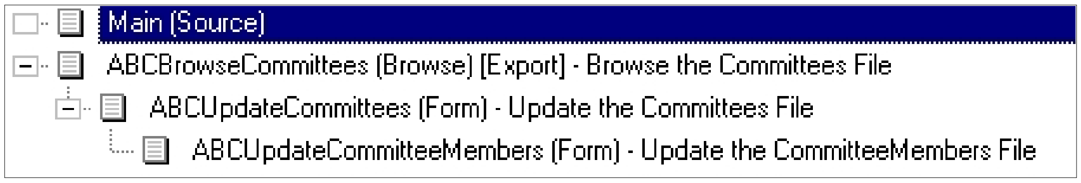
As I said in part one, the method I'm describing can be used to both convert and extend your application with ABC. The first task you'll be doing is adding a new ABC procedure and calling it from the Legacy app. In fact, you'll add a both Browse and Form and call the Browse from the Legacy app.

Follow these steps:

1. From the main Clarion menu select **Procedure**, **New** and name the new procedure `ABCBrowseCommittees`.
2. Select **Procedure Type** of *Browse* and ensure the **Procedure Wizard** check box is on. Ignore the Wizatron option in the following option box if it shows.
3. Follow the Browse wizard selecting the Committees file, name the update procedure as `ABCUpdateCommittees`, de-select the **Provide Buttons for Child Files** check-box and accept all other defaults.
4. Because of the relationship between the Committees file and the CommitteeMembers file, you'll actually end up with an extra Form procedure named `UpdateCommitteeMembers`. Rename this to `ABCUpdateCommitteeMembers`.
5. On the **Procedure Properties** window of the two forms, turn off **the Export Procedure** check boxes.

The final application tree should look like that shown in Figure 4. Make this new DLL.

Figure 4. ABC Procedure DLL with new ABC Browse & Form Procedures.


```
Main (Source)
ABCBrowseCommittees (Browse) [Export] - Browse the Committees File
  ABCUpdateCommittees (Form) - Update the Committees File
    ABCUpdateCommitteeMembers (Form) - Update the CommitteeMembers File
```

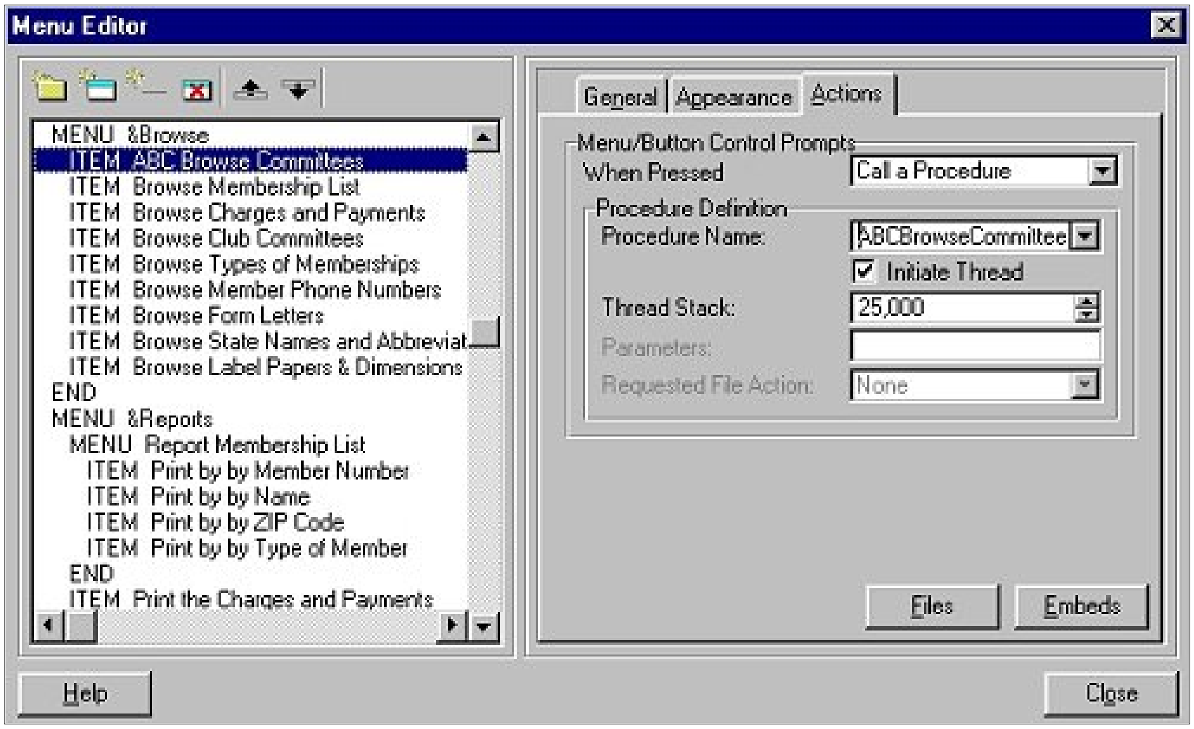
Calling the New ABC Procedure

Now to call the new ABCBrowseCommittees Browse from the Legacy app. Open up the Legacy app created in part one and perform the following steps:

1. Go into the **Menu Editor** on the *Main* (Application Frame) window and add an item to the *Browse* menu entitled *ABC Browse Committees*. Set this item to call a procedure named ABCBrowseCommittees (Figure 5).
2. Save the menu and your Application Tree should now show the procedure ABCBrowseCommittees as a ToDo procedure (Figure 6).
3. From the **Application** section of the main Clarion menu, select **Insert Module**, choose **External DLL** and insert the ABC Procedure DLL library name for the DLL you just created (Figure 7).

4. Double click on the ABCBrowseCommittes procedure and choose **External** from the selection list. On the **Procedure Properties** window, choose the library you inserted in (3) from the drop-down **Module Names** list (Figure 8).

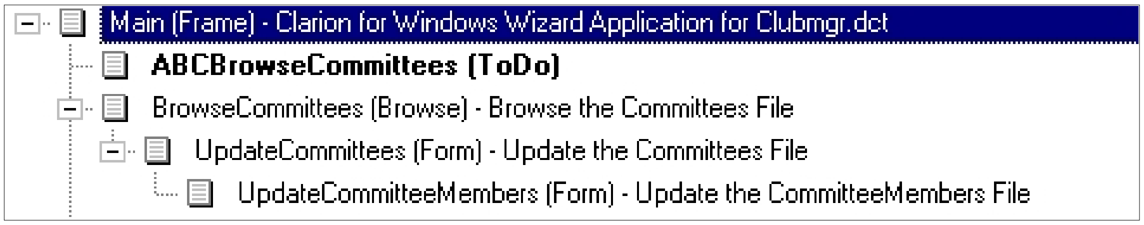
Figure 5. Adding menu item to Main procedure.



```
MENU  &Browse
  ITEM  ABC Browse Committees
  ITEM  Browse Membership List
  ITEM  Browse Charges and Payments
  ITEM  Browse Club Committees
  ITEM  Browse Types of Memberships
  ITEM  Browse Member Phone Numbers
  ITEM  Browse Form Letters
  ITEM  Browse State Names and Abbreviat
  ITEM  Browse Label Papers & Dimensions
END
MENU  &Reports
  MENU  Report Membership List
    ITEM  Print by by Member Number
    ITEM  Print by by Name
    ITEM  Print by by ZIP Code
    ITEM  Print by by Type of Member
  END
  ITEM  Print the Charges and Payments
```

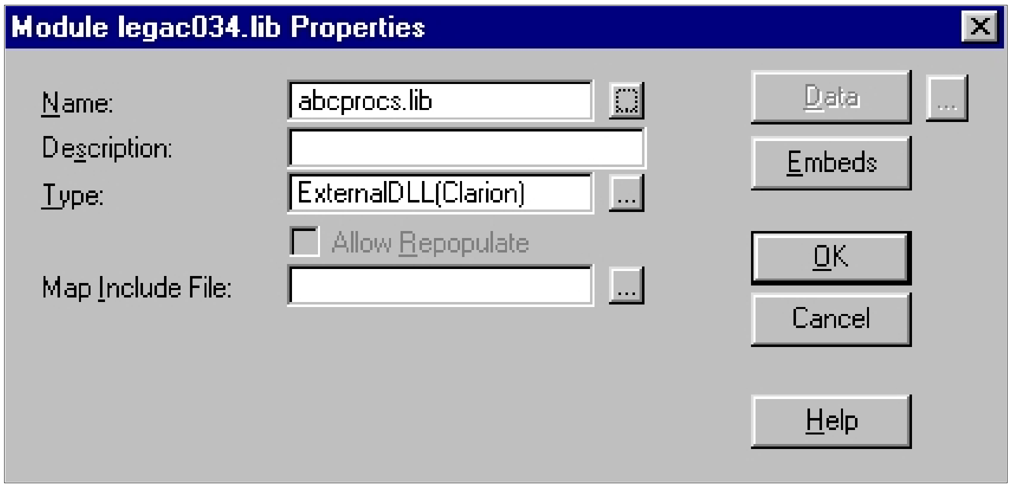
Menu/Button Control Prompts
When Pressed: Call a Procedure
Procedure Definition
Procedure Name: ABCBrowseCommittee
☑ Initiate Thread
Thread Stack: 25,000
Parameters:
Requested File Action: None

Figure 6. Application Tree showing new ABC procedure.



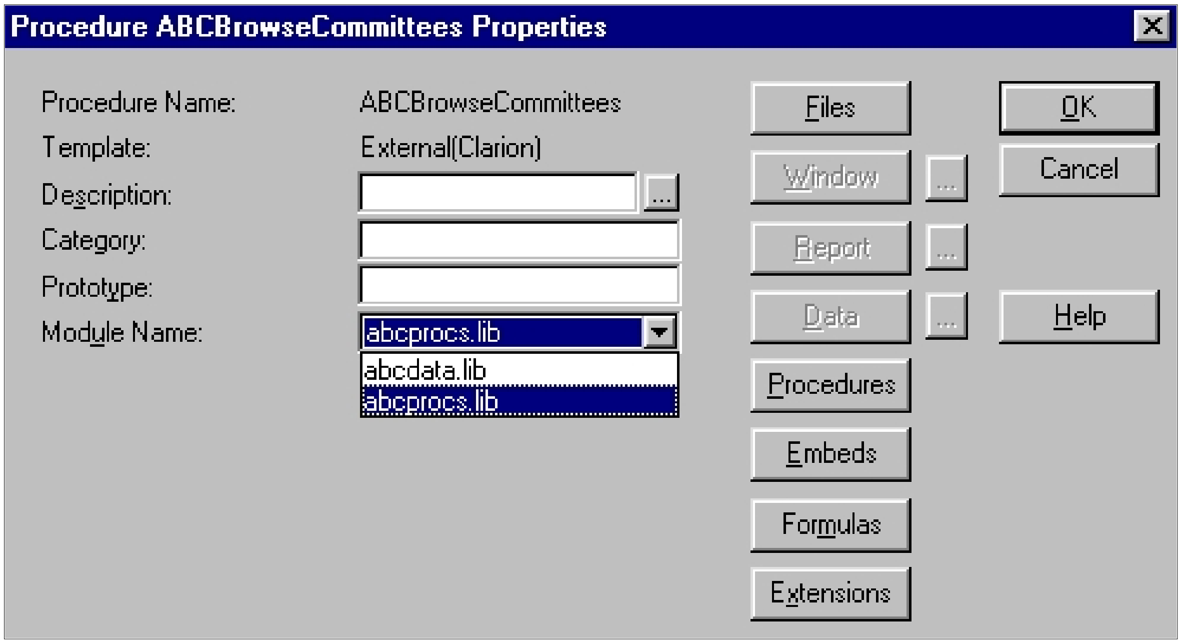
```
Main (Frame) - Clarion for Windows Wizard Application for Clubmgr.dct
  ABCBrowseCommittees (ToDo)
  BrowseCommittees (Browse) - Browse the Committees File
    UpdateCommittees (Form) - Update the Committees File
      UpdateCommitteeMembers (Form) - Update the CommitteeMembers File
```

Figure 7. Inserting ABC Procedure module.



Name: abcprocs.lib
Description:
Type: ExternalDLL(Clarion)
☐ Allow Repopulate
Map Include File:

Figure 8. Choosing ABC Procedure library from Procedure Properties dialogue.


Procedure ABCBrowseCommittees Properties

Procedure Name: ABCBrowseCommittees
Template: External(Clarion)
Description:
Category:
Prototype:
Module Name: abcprocs.lib
abcdata.lib
abcprocs.lib

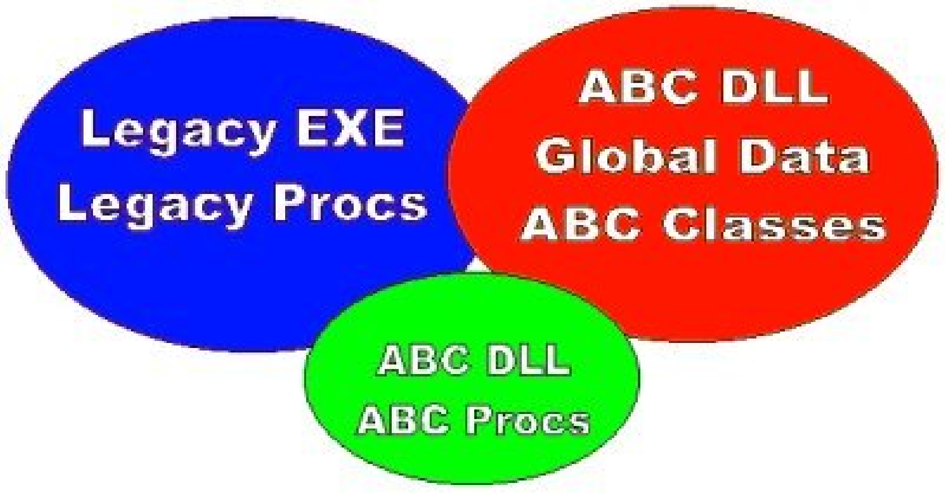
That’s it. **Make and Run** the Legacy EXE and try out the ABC Browse Committees browse. You should find it works just fine! Try using the forms etc.

Add Some ABC Functionality

To make life more interesting, go back into the new ABC Procedure app and turn on Edit-In-Place or make some other ABC specific changes. Re-make the DLL and run the EXE again. This should also work just fine. Please experiment as much as you wish at this stage.

Now, hark back to the diagrammatic representations of your app shown in Part One, and compare it with Figure 9, which is what your app looks like now. You can clearly see how the ABC component is growing.

Figure 9. Hybrid Legacy/ABC app.



No apologies for making this tutorial quite long-winded. In reality you’ve probably done just 40 mouse clicks to get to this stage, but I’ve deliberately been taking this tutorial slowly so I leave nothing to chance – I want you to really understand what’s going on. Also, no apologies for not including any source code to date because I feel you really need to perform the tutorial to grasp this conversion method.

Wow, it looks like I'm out of time again. I was planning to go on and show you how to convert your existing Legacy procedures to ABC in this part of the article, but I see that I've included quite enough for you to get comfortable with for now. Stay tuned for next week's enthralling finale – conversion or bust!

Simon Brewer is Software Development Manager for First Ecom, an Internet development company using Clarion. Prior to that he spent 17 years at Email Major Appliances, major Clarion users and Australia's largest manufacturer of whitegoods. In his spare time he is also the President of the South Australian Clarion User Group and a co-organiser of the ConVic conferences.