CoveComm Inc

Clarion magazine

Volume 1, Number 2 - March 1999

Issue Index

Vol 1, No 2
March 1999

Feature Article

Snortaziw

Wizards from back to front

by David Bayliss

Better is the end of a thing than the beginning thereof: and the patient in spirit is better than the proud in spirit. Ecclesiastes 7:8

We live in an age where speed is often valued over precision. First impressions count, opportunities only come once. We are also victims of whim and fashion. Selling is about working within a market window. The trend is your friend. You have to jump on the bandwagon or you might get stuck in a rut.

The preacher, a great investigator and observer of his day looked upon all that is done under the sun and said, the end is actually better than the beginning. Short-termism is not really the way forward: you need to take the long view. A particular trick or method may work for now but what are the implications?

Then the preacher goes on to identify one of the driving forces behind the reason we want things quickly: it suits our ego. I did this and that and I'm only 25. I got this working in only 3 days. Well, says the preacher, you can do it all that way, but far better to pace yourself, work hard and wait for the reward to come which will be better and longer lasting.

Of course the bible is concerned with matters spiritual and *long-term* can easily mean eternity. I don't want to stretch *quite* that far in this article but I do want to take a long-range look at the issue of productivity. Specifically I want to look at the day-by-day role of the Wizatrons in increasing your work-rate.

Backwards it do to want also I.

The hardest thing I have found about explaining Wizatrons to people is that there is so much depth and detail that MEGO (my eyes glaze over) has set in before I hit the best bit. So for this article I'm going to break all rules of logic, decency and education and structure my argument backwards. Most of the terms you see and techniques I detail will appear from thin air without any preparation or warning. Don't worry. There are many people writing excellent 'how to' articles (including our documentation department); I'm trying to provide the framework for it all to slot into place.

Huh! Another article with loads of hype and no action! Possibly. I'm trying to give you a glimpse of the 'end' to encourage you through the 'beginning.' And there will be a beginning. The Wizatrons are the ultimate in the Bayliss notion of the 'up front hit,' work harder today so that next week is easier. The Wizatrons will work straight out of the box, but the *real* benefits will come to those who work their work patterns around keeping their Wizatrons up to speed.

What do I do and how do I stop?

The key to understanding the Wizatrons' astonishing claims is that they are designed to work with (not instead of) the ABC templates. So first consider how you can use the ABC Templates to optimise your work pattern. Assume you're generating an arbitrary update form.

1. Wizard up the form to get the fields kind of right.
2. Jiggle the screen a little to make it look right (including lookups etc).
3. Consider the special functionality required on this form, select from your toolbox of 3rd party (and self written) templates to add control templates/procedure extensions as appropriate
4. Fill in any template prompts with special values as required
5. Add hand-code to embeds. Test/debug ad nauseum

I have hopefully covered all the bases for an app-gen style app. Ignoring Wizatrons momentarily, how much time do you spend on 5? If the answer is 'a lot' then this suggests you need to enhance your box of 3rd party tools or that you need to get into the habit of writing templates and/or extending the base classes. If you spend your time in 5 then the Wizatrons aren't going to help you much, but equally you are already missing most of the power the IDE offers. The ABC and templates offer you amazing code re-use and are the corner-stone of RAD. But hang on! Case5 is where I add my value to the client! But is it? I would suggest any code you write falls into one of these categories :

. You're tackling the same task as a 3rd party but not quite so completely. In other words if you used the 3rd party tool your customer would get more value. QED.

b. You're tackling the same task as a 3rd party tool but doing it better. In which case you have a better quality general purpose tool so your code should be embodied in a wrapper around (or instead of the 3rd party tool). In other words this can be viewed as a long term case 3.

c. You're tackling a problem that uses your specialist knowledge to write code or an algorithm that a general 3rd party couldn't tackle. So use your knowledge to come up with a general solution to that problem, wrap it in a class/template and again you have case 3. You then have three potential gains to this work: I) You could sell your solution to other developers; II) You can demonstrate to future clients your ability to solve this problem *very* fast; III) if ever you expand your operation (or sell it) you have demonstrable value in your code base.

d. You are doing something completely unique that you'll never do again... Why? If something is worth doing once it is worth doing twice (assuming you stay in a similar area of work for reasonable periods). If it isn't worth doing once then why are you doing it?

Viewed this way the powerhouse of the development process becomes step 3, shoving all the control templates and extensions into the procedure.

So how do you get the Wizatrons to do case 3 for you? Well, suppose you want to populate the UUUU extension onto all your forms (for evermore). You go to the form Wizatron (in the configuration utility), select new property, give it a name, and say it is an *acorn* that populates the UUUU Wizatron. There you go, the UUUU extension will be on your forms for evermore. (Forward references remember – *Cheat: An acorn is a tree waiting to happen*).

Prejudice

But hang on, the problem is you don't want the UUUU extension on *every* form, only *some* of them. Using 3rd party (or your own) templates isn't just about owning them, it is about knowing when to use them. So, the next step is to teach your Wizatron when to apply the given extension template. Now, if you ask someone why they populated a given extension you will typically get one of four answers :-

a) I always do it for this kind of procedure. There is something about the procedure itself, or perhaps the other templates in the procedure, that causes you to decide to populate a given extension. For example, you populate the resize extension if the window has the resize attribute. Enter wizatron macros. When you define an acorn you can also define an expression and the Wizatron will only actually be created if that macro evaluates to true. Every property that every Wizatron has is available from the macro language. For instance

```
@EVAL(CHOOSE(UPPER('%FrameStyle%')='RESIZE'))
```

is the expression that decides whether or not to populate the resize extension.

```
@EVAL(CHOOSE('%LookupFile%' AND '%LookupStyle%'='Combo'))
```

is used to decide whether to populate a file drop combo template to act as a lookup on a field that is related by a many-to-one relation.

b) I always do it for this kind of file/field. There is something about the file, or field (or key etc) that means you want to populate this particular control template or extension. This brings us to a tremendous Clarion strength, the data dictionary. The Clarion dictionary has about three times the number of properties per field over a standard SQL data dictionary, and the interpretation of those properties is the main trick to making the templates look so smart. Now the Wizatrons take this a step further with a function @USER(myfile.myfield,smells,1). This little function tells us the value of the smells attribute of a given field. The 'smells' can, of course, be replaced with anything you want. Suddenly the Clarion repository has an infinite number of attributes per field that can be set for any purpose you wish. As an example the field %LookupStyle% used above indicates what method should be used to do a lookup. It (in turn) is actually a macro dependant on the LookupFile.

```
@EVAL(CHOOSE(@USER(%LookupFile%,Small)=1,'Combo','Lookup Button'))
```

In other words, go look in the dictionary, if the user has flagged the file as small (having few records) then populate a drop combo, otherwise use an entry with a lookup button. You can use a similar technique to populate a * next to required fields etc.

c) I always do it for this customer. You could do this by giving yourself a customer property in the application Wizatron (just use new property, display type drop-list, choices Fred,Bill) and then defining all of your properties (including the acorns) in terms of the customer property...

```
@EVAL(CHOOSE('%Customer%'='Fred','DnArrow.ICO','MONICA.ICO'))
```

However, you are probably better off using the multiple style-sheet facility and simply produce a style sheet to suit given (long term) customers.

d) It seemed like a good idea at the time. Yep, well, we all hit this eventually. So the Wizatron allows you to save a arbitrary decisions you made last time, just in case you want to re-run the whole thing again with a few tweaks.

Play

It might be good to digress a little and come at this from a different angle. If you read any computer magazine it won't be long before you read about components, or plug and play. The idea is actually very simple. Rather than working at the lowest level the language can why not build up lumps (components) of functionality? Then you can achieve work by plugging the components together, and then tweaking them. This is the central notion behind JavaBeans and COM.

Now Clarion templates have always supported this notionally. Legacy control templates were really components and the procedure window was a way of plugging them together. The problem was that once the source code came out the components had all been shoved into a pot and whisked up so you were left with soup. ABC was really a bottom-up attack on component ware. It let us actually build components (and give the customer the capability to build components) that can be plugged together and played with at the language level. So when you populate templates in an ABC procedure you are really *plug*ging them in, when you fill in dialogues you are *play*ing. Then, if you look at the source code, you will see a plugged and played procedure.

So, the logic goes, if you have launched a bottom-up attack on componentware how can you achieve the next big thing? Launch a top-down attack. Rather than just building bigger components why not see if we can help with the plugging and playing. Plugging is of course precisely the task I described in 3) and which I showed the Wizatrons could do leaving you to do the playing (item 4).

Except the Wizatrons can do this too. To see how they do this you need to think about what kind of things you actually put in the prompts of the template dialogs. I think the list below is fairly comprehensive.

. References to other components. A number of prompts developers enter are simply there to plug this component to another one. For example, on a Fieldlookupbutton you need to name the entry control you are looking up. Now remember it was the Wizatrons that decided to create this button (they had probably spotted you had an entry field on a foreign key linked to a big file) so they have the information about which entry the button is for, so they fill it in for you. Similarly they know the procedure you want to call (they generated it). They know the linking key (it's in the dct) and they know the field being looked up (they populated it). Similarly the file schematic can simply be slotted into place from information the Wizatron has. So all the structural prompts can be filled in for you.

b. Prompts you wish we had defaulted some other way. There are a whole set of prompts that you almost certainly fill in the same way on every procedure (or at least every procedure of the same type). Resize strategy maybe? Buffered Reads?. Well, each Wizatron has a set of default values that are used when an instance of that Wizatron is created. These are editable in the style file. So any prompts you always fill in the same way, you fill in once in the style and from then on the Wizatron can do it for you.

c. Prompts you fill in based upon values in other prompts and/or the dct. How about Use edit-in-place or the locator type/field. Here the prompt values can be computed from the dct and/or other properties on the procedure. This is similar to case b) of the problem 3 solution. In short, using macros you can often compute satisfactory prompt values. (Remember, you can define arbitrary data attributes in the dct to help your macros work).

d. Prompts you fill in as a set. In other words, prompts whereby once you've typed in the first the next 8 can be predicted. For example, if you say the button text is 'OK' it is reasonable to guess the icon will be a tick, the size will be such and such, default will be set etc. The Wizatrons have a derivation capability so you can define 'common' sets of prompts that are filled in for a given Wizatron. If you like a Wizatron can have many different default sets and you get to choose one.

e. Prompts you felt like filling in to see what they would do. This is case 3 and is again handled by the detail tree.

So you see not only can the Wizatrons plug all the components together, they can actually fill in all the prompts in the components to give you the kind of component you want, working the way you want.

And finally (the bit you've all heard about)

So, we have always had a solution to issue #1. Eliminating #5 comes down to disciplining yourself and is the art of productive CW working anyway. #3 is tackled by the acorns and #4 is tackled primarily by the ability to set families of wizatron defaults and use macros involving data from the dictionary.

This leaves #2 which is the bit we have talked about most because it is the easiest to demonstrate. The solution to 4 allows for useful little tricks such as being able to define an insert button complete with icon, colour, message, tip etc. However to *really* get hold of how the Wizatrons can compose screens as nicely as you can you need to complete a simple test :

You are in screener and have selected all of the buttons near the bottom of the screen. You right click and select 'Align Bottom.' Were you thinking :

i. I want all these buttons to have a ypos of 32; this button on the right is already there so I can quickly move them all to that position by saying align bottom.

ii. I want all of these buttons to be the same size…. Damn why did I click on that? Now I've got to re-hack the entire screen.

iii. I want all of these buttons to share the same ypos and I want it to be along the bottom of the screen with a pleasant margin to the bottom of the window.

If you answered I) then the Wizatron screen composition is not for you. If you answered ii) then the Wizatrons are definitely for you but you need to be a little soberer next time you try to use them. If you answered iii) then you are well on your way to writing your first style sheet.

Seriously, the essence of the Wizatron attack on problem two is the notion of a composition rule. A style sheet says to a Wizatron "I don't care where you shove the controls as long as you obey these rules". What kind of rules? Well there are many but here are some :

. These controls must be aligned on x or y co-ordinates. This is done using at.xpos/ypos properties. You don't use numbers; you just use labels. If two controls share a label then the Wizatron will make sure those values end up the same. You can force widths or heights to be the same using a similar technique.

b. These controls must be in this part of the screen. Each window is divided into the points of the compass and centre. You can specify which zone a control goes in.

c. These controls go on this tab. Tabs are specified by name and created if required. Overflows are automatic.

d. This control should be this size, or failing that … Obviously controls have size, but you can also specify how they squash/expand if required to help fit in with other rules.

e. The space between controls must be so much … You can specify margins, although you can also specify that some controls have more (or less) margin on a control by control basis.

Some people have an almost theological objection to automated screen composition. Aesthetics is subjective and therefore cannot be automated.

I actually half-agree with this. If the task in hand was 'make this window as visually appealing as possible' then I suspect a skilled designer would win most of the time. However, that *isn't* the assignment. The assignment is 'make this window as visually appealing as possible *given* it has to fit in with all my other ones.' To a human the second clause is another burden that makes the task even harder. To a Wizatron it is the solution, it means that consistency and precedent can be used to define most of the screen design using composition rules to fill in the rest.

Wizatrons

So there we go, this is the end of the article. If the opening verse is correct the conclusion should be better than intro, and those of you that have patiently ploughed to the end of the article should have done better than those who proudly ignored it. I hope that it is so.

David Bayliss is a Software Development Manager for Topspeed Corporation. He is also Topspeed's compiler writer and the chief architect of the Application Builder Classes.

Vol 1, No 2
Mar 1, 1999

CoveComm Inc.

Clarion magazine

Feature Article

Interview: Roy Rafalco

A Conversation With Topspeed Corporation's New CEO

Part 2

On December 28, 1998, Topspeed announced the appointment of Roy Rafalco, Topspeed's President and Chief Operating Officer since 1993, to the position of Chief Executive Officer. Rafalco succeeds Bruce Barrington, Topspeed's founder, who will remain Chairman of the Board. Click here for the full press release.

This is part 2 of a two-part interview. Click here to read Part 1.

Clarion Magazine: How are you positioning Clarion in the marketplace?.

Roy Rafalco: It's critical, because you and I both know, once people start using tools, they attach themselves to those tools. It's not easy to change. It's a learning curve, and it's even worse if you're not MS or IBM, and you're some Clarion Software or Topspeed Corporation that you never heard of before. That's why the word of mouth is essential. Right now our marketing department is gathering all sorts of case studies and success stories, so that when we do promote and call on people, we have a success story. People need to hear about other companies that have successfully used Clarion.

We used to hear about the push to sell Clarion in the corporate shops. Is that less of a focus now?

We were attracted to corporations for a couple of reasons. One was the lack of price sensitivity. Secondly, corporates were willing to talk about the tool once they had purchased it. If we could sell into a corporation, chances are they were going to pass the word about Clarion on to others. There's no competition between corporate developers about which tool they used. But the corporate market, which is really client server, is a mature marketplace. And when corporations make such an investment in the tool, the decision-maker's not going to say oh, you remember all those millions of dollars I spent on PowerBuilder? I'm going to spend a couple of thousand dollars on Clarion and do the same thing. It doesn't work very well.

All along our goal has been the same.We want to sell to programmers who can benefit from our technology, whether it's a small shop or a large shop. Now, we will be targeting and continuing to want to target programmers where the cost of selling the tool is reasonable given our produce price. Since, we don't charge royalties or runtimes, we receive only one price, and that's it. We may not ever hear from the customer again, unless they want to upgrade. Due the cost of selling the tool, it makes no sense to knock on the door with one or two programmers in that organization.

So if not corporate shops, what's the market?

One area that we're thinking about carefully is the trend in the marketplace is to outsource IT. We believe IT consulting firms are less married to their tools. IT consulting firms have more programmers who may not be committed to a particular tool. And that's why there's some significance in the press release you might have seen on UTA, where they've decided to train their programmers on Clarion. They're a government consulting firm. UTA's interest in Claron is significant when you consider that their client is very conservative.

We still want to sell to programmers, and we don't care whether they're small independents or they're large corporates. Eventually, we will better identify specific markets where our tool will provide an unique solution, and our cost of sale will be reasonable.

What about creating a version of Clarion for Linux? Is that something you're keeping an eye on?

Oh absolutely. We've certainly learned our lesson when it came to getting to Windows late and getting to Client/server late. Being there first means a lot. So we're constantly thinking about that, and meetings go on every week, where we're talking two, three, four, five, ten years out. What is the latest technolgoy wave, and do we grab that wave or don't we grab that wave? We had a painful lesson back in '93 when PDAs were the big thing. You remember the Pen Developer add-on to Clarion 3. I think we may have sold two of those. Because we have limited resources, we have to be careful where we devote our development resources

We all at TopSpeed have worked too hard to throw things away because we went in the wrong product direction. We have retained the Gartner Group as analysts for us. They provide consulting and services in terms of where they see the direction of technology going, and the marketplace. We consult with them regularly. Yesterday Bruce brought to me the fact that Infoworld estimated there was 12 million Linux boxes out there.

As we come to the right business decision, we also have to decide who we can link up with strategically to make it happen. Because you can't do everything by yourself, we have to constantly think about matching ourselves with a partner.

Will there be an IPO any time soon?

We don't want to go public until we think we can really maximize the initial stock offering. And we believe that that will take several years of consistent growth and we have to be at a certain revenue target. It could be as early as two years from now.

Are you thinking at all about the Open Source movement? Would you consider opening up the Clarion language?

Not really. We've been reading about open source, and it depends on your viewpoint. We've seen some things take place that look more like gestures than anything else, but on the Clarion side, we really don't see a need. Our plans are to continue advancing what we call the art of automated programming to whatever platform, using whatever language. At the same time, I would never say never to anything, but we are not ready to open up the Clarion language at this time.

How about generating Java or C++ code?

Given our technology and the fact that you don't have to hand code as much, the language used ends up being irrelevant. I can't necessarily tell you which direction we're going to go in terms of supporting other languages, However, we're in the long term business of providing automated programming tools.We'll look at all of the opportunities for how we can leverage our automation technology in order to grow the company and the technology so that programmers can be more productive.

If we were to do another language, Clarion wouldn't go anywhere. We still believe in the Clarion language and we still think it's a good language. When I was in London in August, I was amazed to hear so many London developers taking the side of the Clarion language. Way back when we merged it was Clarion what?

It's also their Clarion language now.

And it's showing. It's a good environment and it's one of the things I really like about Topspeed and the people, and the London guys are just fantastic to work with. They could be so much different, based on other engineers I've worked with.

It seems like there's a lot of loyalty and stability there.

In 1997 I was in London and some of the original JPI people came in to see me. It was the fifth anniversary of the merger. And I thought Oh God, it's five years, how do they really feel about the company. However The London developers said it was the best decision we've ever made. They love what they do, and hope that the company will continues on like it has been.

What do you see as Topspeed's biggest opportunity?

From a technical viewpoint, there is no limitation as to what the London developers can do. I sleep well every night knowing that the London Development Centre exists with its tremendous talent. So that itself is an opportunity. Very few companies have that kind of talent.

I think the biggest opportunity is that we haven't told anyone about Clarion. If we start telling people about Clarion and being focused with our sales and marketing programs, I just think there's an incredible opportunity for the company as a whole. I'm just amazed at what we're able to do and no one's ever heard of us. I was just blown away when I saw this posting that we were second at DBMS. I just think if we come up with the right sales and marketing approach and start this ball rolling down the hill, then we will have an expanding the number of customers. When we increase our install base, our financial resources increase. With increased financial resources, we can enhance the technology even more, and it just goes on and on and on.

As I was coming up on my fifth anniversary at the end of 1996,I made a new year's resolution. I've been here five years. This company's going to go, if I gotta kick it, I gotta shove it, I gotta whatever, but I'm going to be less patient now. I'm a very patient person, very methodical., Over the years some of my staff have criticized me because I'm too patient. I should've made some moves quicker, whether it's regarding personnel or whatever. And so, I've got my staff working based on this resolution. I've told them they've got eight quarters. In eight quarters, we're going public, and we deserve it! I mean I'm not going to sit here and drive the car off the cliff, but this is our time to shine.

Remember Internet Connect? A couple of years ago I was over in London and I was sitting down with the guys. At the time, we were pushing internet connect to be mastered. Gavin Halliday was giving a demonstration of a very rough alpha and showing what it could do.

I was sitting there with my mouth wide open in amazement. I turned to Gavin and asked him how he created such amazing technology.

And Gavin looked at me quite matter of factly and said, "Well Roy, it's magic."

And all I said was "OK, Gavin, I guess you've decided I'm not technical enough to understand. Just pretend it's magic, right?" And he said "Yeah, it's just magic."

Your press release says that your appointment confirms what you were already doing. Did Bruce just sort of look around and say, well, he's acting like a CEO, I guess we better promote him? How has your role changed?

Bruce doesn't necessarily turns over his favourite thing to anybody, and Bruce is very cautious. Over a period of time he's gradually given me more responsibility. He came to the conclusion that now basically I was running the business, and it was time for me to come from behind the curtain. And quite frankly it wasn't my idea and I didn't ask for this, because it's been really comfortable to stay behind the scenes. I can hide out back there and make things happen, and not necessarily deal with all of the political things that result from being in the limelight.

As things are getting better at TopSpeed and we're breaking all records, I turn to the old folks here (they're not that old, but have been with the company), and I say you know, we're going to miss those old days, because a crisis brings people closer together towards a common goal better than anything else.

It's also probably true to some extent of the programmers. If it really got big, one of the things that we could lose would be that close familiarity.

Customers tell me that they don't want us to grow big, because they know they can get me on the phone. Also, they know they can send an email, and Bruce may respond to it. They don't want those days to come to an end. The tool doesn't cost them too much. They like that kind of premiere access to TopSpeed. In addition, their competitors do not knows about Clarion. so they can go in and undercut their competitors' bids.

Perhaps some of the lack of growth of the product is because of that resistance to change within the Clarion programming community.

My plan is to continue to make it that personal touch, even though it's going to be hard, and it's going to get harder and harder. But that's a value that our customers like. And we'll strive to keep it as close as we can.

Clarion wouldn't be where it is today without its customers. And those customers are a different kind of customers. They're the customers who pay for their upgrades every year and kept us in business. They're the customers like Team TopSpeed who are just giving of their time to the nth degree to help everybody, and to help the company. Really the success here is all based on the customer. That's our greatest asset.

Vol 1, No 2
March 1999

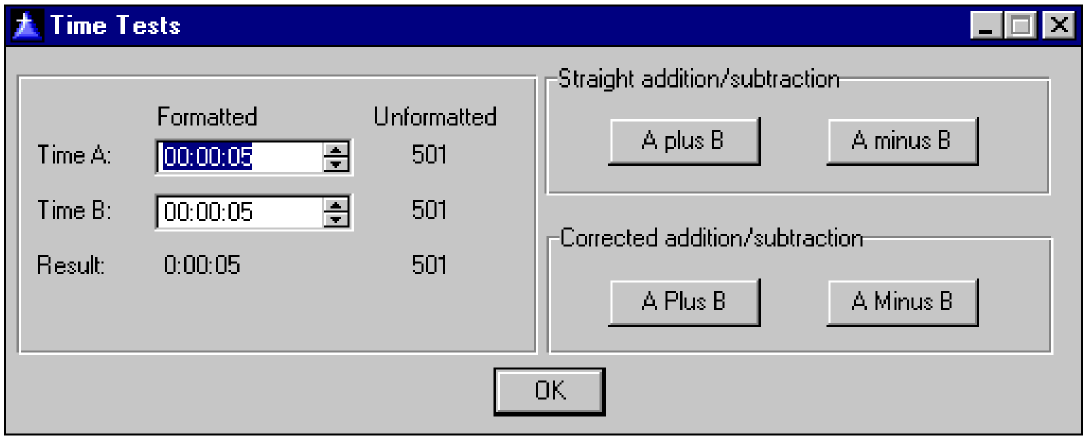
The Clarion Advisor

The Mathematics of Time

If you've ever needed to perform math on Clarion time values, you know that you can sometimes get unexpected results. Clarion stores times as hundredths of seconds since midnight, which means that one second is equal to 100. Given this you'd expect that five seconds after midnight would be stored as 500, but in fact it's stored as 501, because a time of 1 is equal to midnight. This extra 1 causes all sorts of interesting problems.

Consider what happens if you try to subtract one time from another time. Figure 1 shows a test application (available for download) which lets you enter two times using the time picture @t4, which lets you enter hh:mm:ss.

Figure 1: The Time Calculation Application



	Formatted	Unformatted
Time A:	00:00:05	501
Time B:	00:00:05	501
Result:	0:00:05	501

If you enter a time of 0:00:05 for the Time A, and 0:00:01 for the Time B, and click on the Unadjusted Subtract A From B button, you get a result of not 4 seconds, as you'd expect, but 3 seconds!

This is *not* a bug, but a failure on the part of the developer (that's me) to account for the one one-hundredth of a second. The time 0:00:05 is stored as 501, and 0:00:01 is stored as 101. 501-101 = 400, and the display picture in effect is rounding down the time value, since it only displays whole seconds. All times from 301-400 will display as three seconds, from 401-500 as four seconds, and so forth.

If you're adding times you may not notice anything wrong, since 501+101=602, and anything from 601 to 700 is going to display as 6 seconds, but if you're doing repeated ads the error will accumulate and eventually could wrap around, adding a second.

One way to solve this problem is to subtract the extra 1 from the times before doing any math, and add the 1 back to the result of the calculation. This way you can do addition, subtraction, multiplication and division and get accurate results. You should also, however be wary of running over the end of day boundary. There are 8,640,000 hundredths of a second in a day, and a time of 8,640,000 is equivalent to 23:59:59:99 (keeping in mind that there is no Clarion time picture that will display those hundredths of a second for you).

```
TimeResult = (TimeA – 1) + (TimeB – 1) + 1
```

This code could also be written as

`TimeResult = TimeA + TimeB - 1`

However, if you're dealing with a long list of numbers, or mixing addition and substraction and possibly multiplication and division, you'll be far better off subtracting the 1 from each time right away, then doing the math, then adding the 1 back.

If you’re going to be adding times that may go over day boundaries, you can't simply add the times, because Clarion won't display time values greater than 8640000. You can either subtract 8640000 from such a time, or do it the easy way with with the modulus operator which returns the remainder after division.

```
CurrTime = CurrTime % 8640000
```

The above code will be correct if you’re adding times, although you may want to keep track of how many days are involved. You can get the number of days with this code:

```
Days = INT(CurrTime/8640000)
```

If you’re subtracting times you may end up with a value that’s less than 1. Here you have to keep in mind that there are only 8640000 hundreths of a second in a day. If you subtract ten seconds from today at 12:00:05 a.m., the result you want is 11:59:55 p.m. yesterday. If the resulting time is less than 1, you need to add 8640000.

```
TimeResult = (TimeA – 1) - (TimeB – 1) + 1
IF TimeResult < 1 THEN TimeResult += 8640000.
```

Clarion times aren't overly difficult to calculate. The most important factor to keep in mind is that all whole seconds are offset from 1, not from 0.

Download the source:
ftp://www.clarionmag.com/pub/clarionmag/v1n2/calctime.zip

Vol 1, No 2
March 1999

Feature Article

Name() Comes of Age

by Steve Parker

Various kinds of files, such as calendars, schedules, ASCII files, accounts, or files on other drives or other machines share the characteristic that multiple physical disk files can have the same structure. It is often desirable to have a single dictionary specification to declare that structure so that a single procedure set (application) can operate on them.

Clarion builds in this capability with the Name attribute but provides no management capabilities. It is up to the developer to manage which file is in process at any given time.

Of Syntax and Semantics

In Clarion, there are two keywords you must not confuse, Label and Name.

Variables and data structures (which include windows, queues, views and files) are referred to by their Label. Clarion code *always* uses the Label.

Name is an attribute. Variables and structures may or may not have a Name; in fact, they usually do not. The *contents* of the Name attribute, if any, are for use of an external object. This could be another program, like a database engine or the operating system, or a Clarion DLL, like a file driver.

In the case of data files, the Name attribute contains the fully qualified directory entry for the file. For example, a file declared as:

```
Customer File,Driver('Topspeed'),Name('foobar')
```

Will look for or create a disk file called foobar.tps in the current directory. Your code will never contain this, only "Customer."

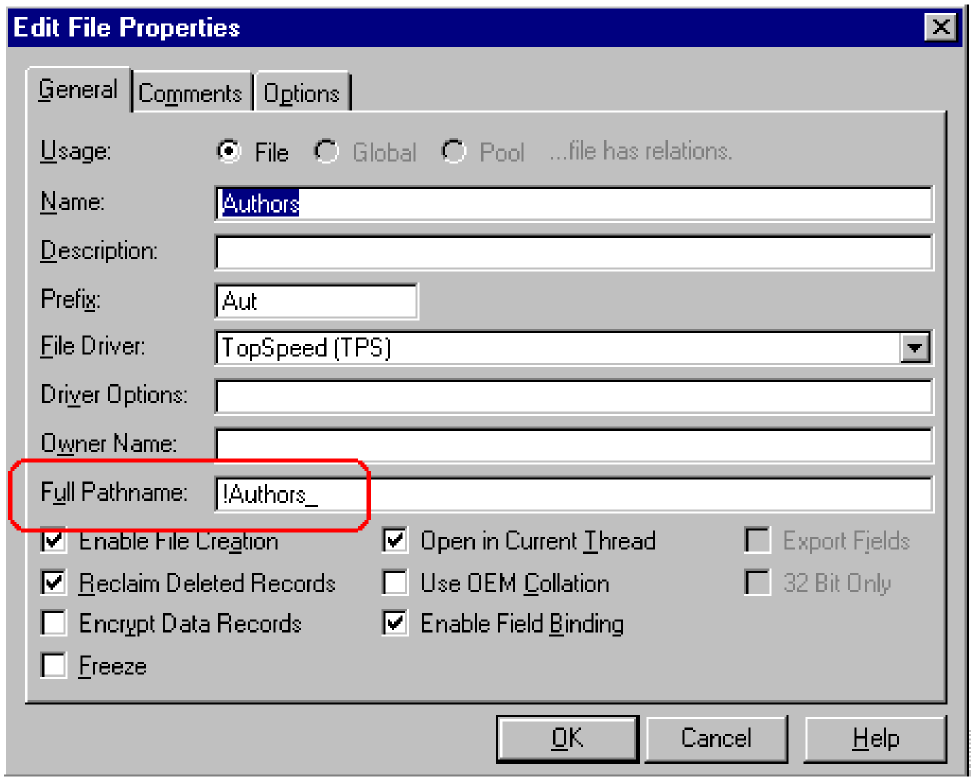
Multiple Files, One Declaration

To handle multiple physical files from a single logical declaration use a variable in the Name attribute:

```
Customer File,Driver('Topspeed'),Name(GLO:FileName)
```

You create this in the Dictionary Editor by completing the Full Pathname field on the File Properties worksheet (Figure 1) and pre-pending an exclamation point.

Figure 1: Global Data Properties



Usage: [✓ File] Global Pool ...file has relations.

Name: Authors

Description:

Prefix: Aut

File Driver: TopSpeed (TPS)

Driver Options:

Owner Name:

Full Pathname: !Authors_

- [✓] Enable File Creation
- [✓] Reclaim Deleted Records
- [ ] Encrypt Data Records
- [ ] Freeze
- [✓] Open in Current Thread
- [ ] Use OEM Collation
- [✓] Enable Field Binding
- [ ] Export Fields
- [ ] 32 Bit Only

This is where the fun starts.

If you declare a file like that in Figure 1, your app will not compile. The compiler error makes the reason clear: the variable was not declared. Not only must you declare the variable, you must do so *before* the file declaration. That means that you have to use the Before File Declarations source embed because the Global Data button places variable declarations *after* the file declarations.

Thus, global files (files in the dictionary) require a global variable. Files declared at the module or procedure level, in hand code, require only that the variable have the STATIC attribute (though it can be at a higher scoping level).

Next, having added the STATIC attribute, if you try to run the app, it will crash and burn. The variable is empty and the app does not know the DOS name of the file it is supposed to open

```
File() could not be opened. Error: Invalid Filename(45).
```

You may initialize the variable any time you wish but obviously not later than the first attempt to open the file.

SetName

The ABC FileManager class also provides the SetName method to set the value of the `Name` attribute. This method allows you to set the variable directly, without an assignment. Moreover, you can call this method in a procedure which does not actually use the file(s) in question (it's a property of the FileManager, hence available any time after the FileManager has been instantiated). The app frame comes to mind, as does the global Procedure Setup embed.

Citing advantages of SetName, Pierre Tremblay observes (in the OOP newsgroup):

> The variable in the name attribute for the file structure doesn't need to be exported [from the data app]. The FileManager class is holding a reference for that var.
>
> So, to "initilialise" this variable without having it exported from the DLL, the SetName method is the only way to go.

Note what Pierre says: the variable "doesn't need to be exported." Yes, this means that you do not need to declare the variable in the non-data app(s).

Rules for Name-ing

1. Always use the file Label as declared on the File Properties worksheet in Clarion language statements;
2. When using a variable in the Name attribute, initialize it before any attempt to open the file;
3. A variable used as a file Name may contain any O/S-valid string;
4. To use a variable, pre-pend the Full Pathname entry on the File Properties worksheet with an exclamation point;
5. The variable name must be unique (of course);
6. Declare the variable in your data application, before the file declaration.

Super Files

Topspeed files support a special syntax to allow multiple tables to be stored in a single file.

By using the special escape sequence '\!' in the NAME() attribute of a TopSpeed file declaration, you can specify that a single .TPS file will store more than one table. (C4 LRM)

> When using the TopSpeed driver, if you wish to store multiple tables in a single physical file, separate the file and table names with "\!," as in TUTORIAL\!ORDERS. This refers to the ORDERS table in the TUTORIAL.TPS file. (C5 On-line Help)

The format of the Name entry is *file_name\!table_name* which appears to preclude using a variable in the Name attribute of a TopSpeed file.

Appearances can be deceiving, for this is not the case. Designate your variable on the File Properties worksheet and initialize it in exactly the same way as any other Name, but use the special syntax:

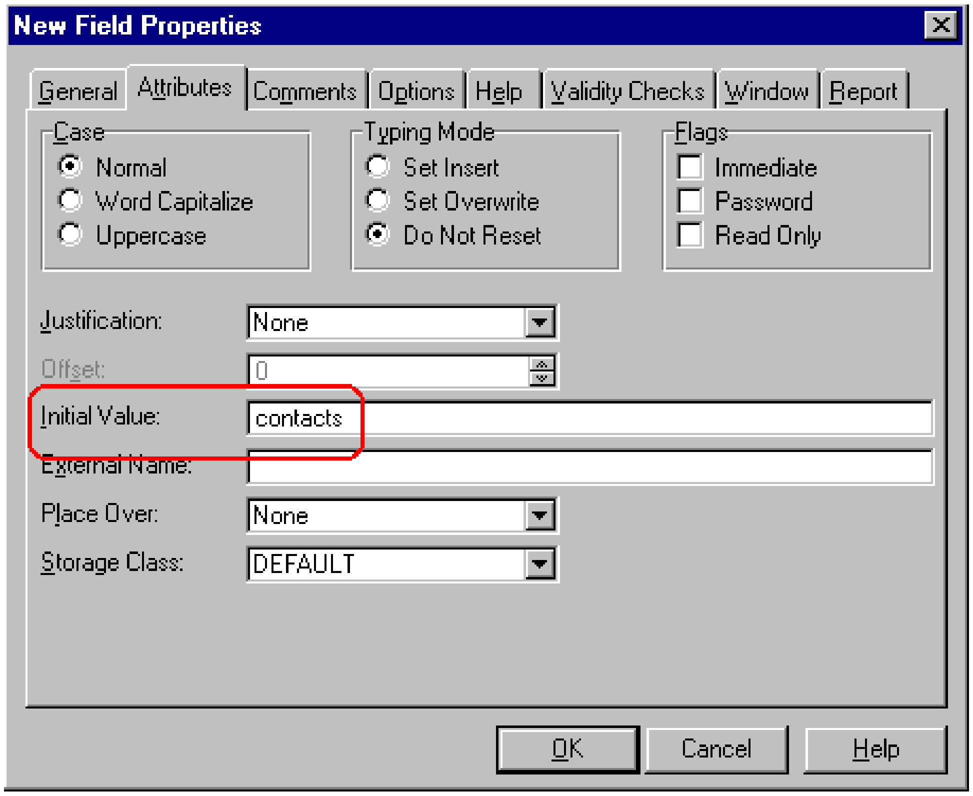
```
Authors_ = 'cwjfil\!Authors'
```

to initialize the variable. In this case, Authors_ refers to the Authors table in CWJFIL.TPS.

Multiple Locations

The executable resides in one place but you need the ability to address files anywhere. The file might be in the current directory or in another directory. The files might be on a network drive. You can even map an IP address to a drive letter and use the Internet.

Figure 2: Setting an initial value for a file name variable.



New Field Properties

General | Attributes | Comments | Options | Help | Validity Checks | Window | Report

Case: [✓ Normal] Word Capitalize / Uppercase

Typing Mode: Set Insert / Set Overwrite / [✓ Do Not Reset]

Flags: ☐ Immediate ☐ Password ☐ Read Only

Justification: None

Offset: 0

Initial Value: contacts

External Name:

Place Over: None

Storage Class: DEFAULT

Here, there is one file (set) but multiple possible paths. In these cases, the path tends to be variable while the actual name of the file tends to be a constant. If so, it makes sense to give the Name variable an initial value when it is declared (Figure 2). If you can get the path information separately, from an INI or configuration file or from FileDialog, just concatenate the two.

```
FilePath = GetIni('JtMatch','FilePath',,'.\JOBTRAK.INI')
Contacts_ = Clip(FilePath) & Clip(Contacts_)
```

Multiple Files

You want to work on one company's accounts then another's. Or you need to update Alice's calendar then Bill's.

You do not have to leave the browse, if you do not wish to (though, obviously, you can and this would be easier to program). All you need to remember is that you must close the current file before selecting the new file. Select the new file, prime the Name variable and refresh the browse.

If you have multiple Tabs and it is your intention that each Tab show a different file in the same browse, you must close, prime and reopen in ThisWindow.TakeNewSelection (ABC) or the ?CurrentTab .. New Selection (Clarion) embed.

Multiple DLLs

Now multiple DLLs are *really* fun if you use variables.

Each variable must be declared in each of the project's apps.

> If you're new to multiple DLLs, you should check the documentation. The important thing is that each app in the project needs to reference the files. But only one should

allocate memory for them. This means that in all apps but one the files are declared and the EXTERNAL attribute added. The one app without the EXTERNAL attribute is usually referred to as the "global data DLL" and usually contains only the file declarations.

In the global data DLL, you simply declare the variable and, if desired, give it an initial value.

In every other app, you must also declare each of the variables. Each variable must also have the External and DLL attributes, similar to:

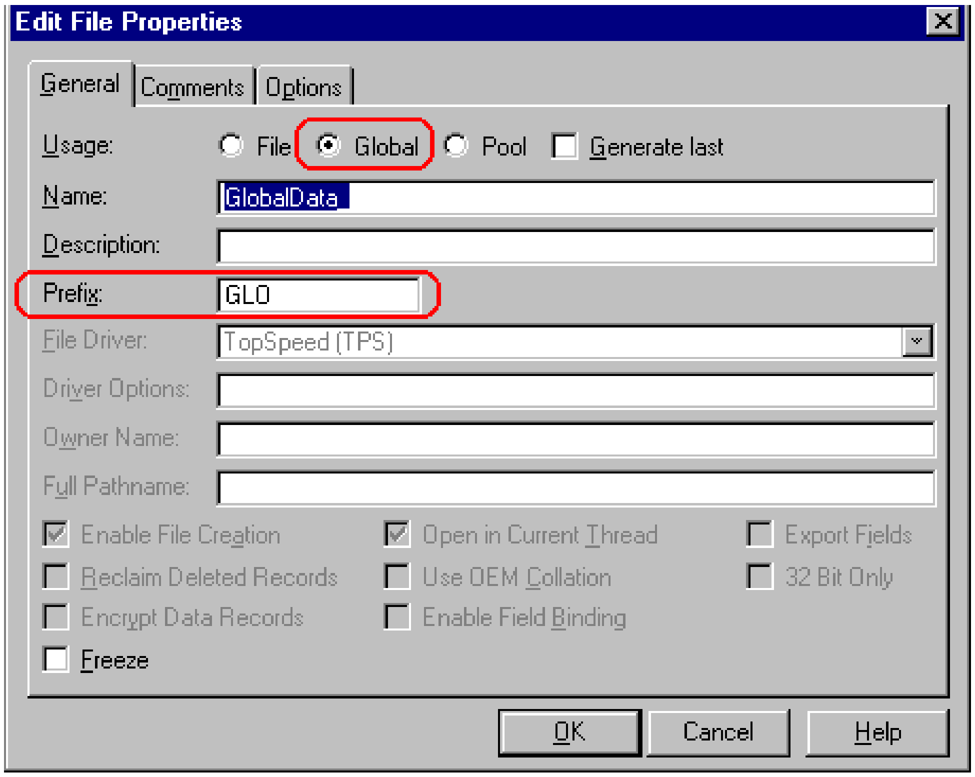
```
Contacts_ String(64),External,DLL(dll_mode)
```

to avoid compiler warnings (lots). (You do not need to do this if you use SetName).

Until C5, the method of choice was to create a text file with the necessary declarations and `Include` it in each application. The required file is usually created by requisitioning and modifying the data declarations from the main source file of the data DLL.

A major enhancement in the C5 Dictionary Editor makes this a thing of the past, totally automating the process.

Figure 3: Dictionary Editor Global Data settings.



Usage: File [✓ Global] Pool ☐ Generate last
Name: GlobalData
Description:
Prefix: GLO
File Driver: TopSpeed (TPS)
Driver Options:
Owner Name:
Full Pathname:
☑ Enable File Creation ☑ Open in Current Thread ☐ Export Fields
☐ Reclaim Deleted Records ☐ Use OEM Collation ☐ 32 Bit Only
☐ Encrypt Data Records ☐ Enable Field Binding
☐ Freeze

Using the new Global option in the Dictionary Editor (see Figure 3) creates a file-like structure accessible in your applications as Global Data. That is, after you declare, say, GlobalData, you insert each variable in the same way you would add file fields. So, from the point of view of the Dictionary Editor this is a file but when accessed within an application, it is handled like global data, as intended.

The only restriction is that if you add several variables to this "file," grouping your Name variables, they will share a common prefix.

However, all of your declarations will be included and included correctly in each and every application. That is, your global application settings for dictionary handling (viz., the external flag) will be picked up and applied correctly without further intervention. (You still have to initialize variables, so *you* still have to do something.)

> *Note: as of C5EE SR1, Arnor Baldvinsson has discovered, this works correctly only with the ABC templates. The requisite template code was not retro-fit in the Clarion template chain. These lines are found in ABProgram.tpw, 218-37 and should be compared to Program.tpw, 90-99. He states that the new lines can be successfully substituted for the old. A copy of the modified template is available at http://www.cwicweb.com/apps/cwlaunch.dll/download.exe.0*

Summary

The Clarion language has supported variable file names since its very beginning. In CDD, support was added to the Dictionary Editor and, partially, to the AppGen. But, until Clarion5, we had to hand code many, if not most, of the required declarations.

"When it rains, it pours." Now you have two methods that are about as automated as you can get. How do you choose?-This is the most difficult possible choice: two solid, reliable and easy to use techniques.

Download the demo application.

Steve Parker started his professional life as a Philosopher but now tries to imitate a Clarion developer. A former SCCA competitor, he has been known to adjust other competitor's right side mirrors -- while on the track (but only while accelerating). Steve has been writing on Clarion since 1993.

COSP Information

The COSP License (draft)
Some concerns have been raised over the use of existing open source agreements to cover Clarion-related products. Here's the current draft of the proposed Development Open Source License.

COSP Articles/Information
General help on open source concepts and (soon) specific articles about open source products.

Product Information
Information about the products that make up the COSP distribution, including product features, bug reports, desired features, and ways to contribute.

Interested In Contributing?
See our open source contribution guidelines.

Questions?
Bring them up in the opensource newsgroups on the Clarion Magazine news server, or email cosp@clarionmag.com.

Downloads

Programming Tools

The CCI Debugging and Profiling Classes

The Clarion Open Source Project

Debugging/Profiling Classes

by CoveComm Inc.

Usage Documentation

- cciDebugClass
- cciProfilerClass

Bug list/work to be done

- cciDebugClass (not yet available)
- cciProfilerClass (not yet available)

Download version 1.0

NOTE: This document covers version 1.0 of cciDebugClass and cciProfilerClass. If you're using version .9 click here.

cciDebugClass Usage

I wrote cciDebugClass as a supplement to the much-maligned Clarion debugger. Even if I had a good debugger it wouldn't always serve my purposes because a debugger gives you a snapshot of the current state of the program. What I often need is a log of what's happened during program execution.

The cciDebugClass lets me log messages (by calls to the trace method, unless I'm using the cciProfilerClass which will create some log messages automatically) and view the log on screen in a window, in a toolbox, or write the log to a file.

To use cciDebugClass with an application first unzip files ending in .INC and .CLW into the libsrc directory, and files ending in .TP? into the template directory.

Close any open application, choose Setup|Template Registry, and register CCIPROF.TPL. Close the registry.

Open your application. Click on Global, then Extensions, and add the cciProfilerDebuggerClassGlobals extension. On the extension properties select cciDebugClass as the base class. You may also want to check Automatically Add Source Files To Project, although this is NOT recommended when using cciProfilerClass (see below).

You can leave the rest of the settings as they are. This will create a global object called dbg, and will initialize the object at program startup and clean it up at program shutdown. I don't generally advocate short variable names but as this is a global and often-used object I think it's okay.

That's all you need to do! When your application runs a trace file called trace.log will be created. To put messages in this file, call the trace method with your message:

```
dbg.trace('whatever you want to appear in the log file')
```

To view the trace log you can call the ViewTrace() method. If you want to bring up a toolbox on its own thread so you can see messages as they appear, call ShowTraceToolbox().

If you want the trace log to record all of the events that occur in your program, go to the Options tab and check Record All Events. This will generate code into all procedures which have windows to log the events received by the accept loops.

Trace messages are cached so that you're not constantly writing to the disk. If you're getting a GPF somewhere in your program cached messages will be lost. You can set the CacheSize property to 1 so that messages are written out immediately.

cciProfilerClass Usage

Having a debug class was great, but what I really needed was a way to automatically generate a procedure call tree. Happily Clarion provides a mechanism in its profiler hooks (and yes, one of the future applications of this class is execution profiling).

To use the profiler class follow the same instructions as for cciDebugClass but choose cciProfilerClass as the base class.

Next, make sure that Automatically Add Source Files To Project is NOT checked. You have to add the source to the project manually when using profiling because you have to set some compiler pragmas, and if you add them automatically the pragrams will be overwritten the next time you do a global generation. The LINK attribute isn't an option because it doesn't allow you to set pragmas on automatically linked files.

Add cciDebug.clw and cciProf.clw to your application's external file list.

cciProf.clw contains prototypes for two internal Clarion functions: EnterProc and LeaveProc. You need to enable these functions so that the profiler class can use them to automatically log the procedure calls via its parent class, cciDebugClass.

While you're in the project editor, highlight the top line of the project tree (the line that starts "Project:") and click on Properties. Click on the Defines tab and type

    profile

in the text box. Close the properties window. Now in turn highlight cciProf.clw and cciDebug.clw and for each of these files type the text

    profile=>off

in the Defines text box. You have now turned profiling on for the entire application, and off for the two source files, which is essential if you're to avoid an endless recursion when the profiling class or debugging class methods are themselves profiled.

Run the application! It will create a trace log of all of the procedure and method calls. You can call the Trace method to add your own messages, and you can also call the ViewTrace or ShowTraceToolbox methods to see the results of the trace while the program is running.

CoveComm

Clarion magazine

The Clarion Challenge

So you think you can write some tight Clarion code?

Try your hand at this problem. The challenge is to come up with either the smallest or the fastest Clarion code to meet the following requirement:

The Requirement

A recent Clarion Advisor column discussed how to calculate time values. One place this is necessary is in tracking billable time. Write some Clarion code to determine the number of blocks of time in a specified period.

Assume you have two LONG fields, one containing a start time, the other containing an end time. You also have a SHORT field containing a value of from 1 to 60 minutes, and a byte flag indicating if times are to be rounded.

For example, assume that time is billed in 15 minute increments. The time period started at 3:00 p.m. and ended at 3:50 p.m. If the rounding flag is set, the result would be three billable units, since 5 minutes rounds down to zero minutes. If the time period ended at 3:55 p.m., the result would be four billable units since 10 minutes rounds up to 15 minutes. If the rounding flag is off, then both times result in four billable periods, since any partial period is billed as a whole period.

Remember that the second time may be past midnight. You do *not* need to consider the possibility of it being more than one day past midnight.

Send your answers to advisor@clarionmag.com

The most compact and the fastest examples will be posted here, along with any others deemed of special interest.

Vol 1, No 2
March 1999

CoveComm Inc

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ

The Clarion Open Source Project

Updated April 05, 1999

The Clarion Open Source Project (COSP) is an open source distribution of Clarion third party products which is distributed by Clarion Magazine. If you're not familiar with the concept of open source software, please read the Feb 8/99 news feature on COSP.

For a list of COSP products click here (subscribers only)

For an overview of COSP-related information click here (subscribers only)

Key Features Of COSP

- All code is covered under an open source license which prevents anyone from making such code proprietary once it has been placed under that agreement..
- COSP information and developer coordination is available to Clarion Magazine subscribers via the COSP main page and the magazine's private newsgroups.
- COSP code is available to the general public by anonymous ftp at ftp://www.clarionmag.com in the /pub/opensource directory.

Who Contributes To COSP?

Anyone who is interested can contribute to the open source project. Subscribers please refer to the contribution guidelines. Non-subscribers can email cosp@clarionmag.com. Clarion Magazine reserves the final decision as to what open source code it makes available in its distribution.

The key word is *distribution*. Anyone can release any code they wish as open source under any license they choose, and anyone is free to distribute whatever open source they find. Clarion Magazine is providing a particular distribution, or collection, of that source code.

Licensing Issues

The initial plan for COSP was to use the GNU Library General Public License, or LGPL. The LGPL has the benefit of being specific to libraries, whereas most open source agreements cover entire applications. Since COSP is mainly about code and templates, the usual open source agreements aren't going to be of much help. However, certain clauses in the LGPL make it almost impossible to license things like templates. It's also not clear how adept the LGPL is at handling some of the issues raised by code reuse in object-oriented programming.

Accordingly, Clarion Magazine has drafted the Developers' Open Source License, which is designed to address the concerns of Clarion developers.

Additional information about licensing is available on the contribution guidelines page (subscribers only).

Who owns the code?

The code is copyrighted and owned by the original author, who has placed the code under an open source license.

Can I modify open source code for my own use?

Yes. Typically this means that the modifications automatically fall under the license agreement as well.

Does using open source code in my application automatically make my whole application open source?

The Developers' Open Source License specifically states that using open source code in your application does not make your entire application open source.

Do I have to distribute source code?

It's normal to have to either distribute the source or make it available on request. Ultimately open source agreements are about preserving access to code for those that want it, and to prevent unscrupulous individuals from making an expensive proprietary product out of something that is substantially free. Although it may seem like a difficult task to prevent this from happening, if the open source code and its derivatives are freely available, there's no point in someone trying to sell an expensive version of a product when all the important stuff is available for nothing. At the same time if someone has a $5000 product that uses a moderate amount of open source code, that code probably isn't going to help someone else clone that developer's product, since much of the value is in non-open source code.

What products are currently available?

The COSP main page contains links to listings of available products. These pages are available to subscribers only.

When will the proposed Developers' Open Source License be available?

You can use it now, if you like. However you may wish to wait until it's been through a legal review.

Vol 1, No 2
March 1999

CoveComm Inc

Clarion magazine

Feature Article

The ABCs of OOP
Part 1 - OOP Basics

by Dave Harms

The introduction of the Application Builder Classes (ABC) and the corresponding ABC templates are the most significant change to Clarion since the arrival of Clarion for Windows 1.0. It would be inaccurate to say that this change was universally welcomed by developers. A lot of loyal customers wanted to know exactly why they were being asked to undergo a major learning curve from procedural code to object-oriented code.

These concerns are valid. ABC presents two challenges to the Clarion developer. The first is the requirement to know something about object oriented programming (OOP). The second is to learn the uses of the ABC library.

The extent to which you have to meet these challenges depends a lot on your current and anticipated style of development. If you mainly use the templates and don't write a lot of embedded code, then chances are you won't actually notice a lot of disruption to your work, since the AppGen interface builds on what is already familiar. You will see a number of new features and options.

If you write a lot of embedded code or other source code then you will need to learn proportionally more about OOP and ABC. Fortunately the payoffs are considerable. For more on the costs and benefits of moving from legacy to ABC see the article ABC or Legacy – Which Templates Should I Use? (Feb 1999).

In this first article of a series, I'll cover some object-oriented programming basics as applied to the Clarion language through some simple examples. Subsequent articles will expand on these examples and illustrate in microcosm how ABC and the ABC templates work.

Understanding Object-Oriented Programming

There's an apparent paradox to learning object-oriented programming. On the one hand, OOP can be very difficult to grasp. On the other hand many people, when they get the basics in hand, look back and wonder what all the fuss was about.

OOP really isn't that difficult. Once you learn the basics of object-oriented programming you probably won't look back on the experience and think you've come a very long way. It's actually a short trip. But it is a challenging one because it requires a change in how you think about programming. And that change will open up vast new programming possibilities which certainly can be daunting. But even procedural programming is like that. No doubt you've seen procedural code that you wouldn't wish on your fiercest competitor.

Put another way, OOP is a bit like those 3D computer-generated images you find in poster shops. The first time you look at one all you can see is a meaningless fine-grained pattern. You know there's something there but no matter how hard you stare you can't see anything but the pattern. Once you grasp the concept of relaxing your eyes and staring off behind the surface of the poster, the 3D image leaps into view. Ah, you say, that was easy.

Similarly, there are some basic concepts you need to get the hang of before you can see the meaning in OOP.

There are three concepts commonly referred to as the pillars of object-oriented programming. These are encapsulation, inheritance and polymorphism. They can be summarized as follows (keeping in mind that anything that can be put into a nutshell probably belongs there).

Encapsulation

Briefly, *encapsulation* means that a class contains code and data. You may think hey, this is no big deal, everything I do is code and data. And you're not far off the mark. But there's more to this, as you'll see.

Inheritance

You've probably heard enough about OOP to have a bit of an idea of what *inheritance* is about already. Using inheritance you can create some code (a derived class) which automatically contains some existing code (a parent class). It's a tricky way of freeing you from retyping code when you want something that's almost like something you just did, but not quite. When you think of inheritance, think of code reuse.

Polymorphism

Polymorphism is something Clarion has had in one form or another since its inception. When you use OPEN on a file, or on a window, you're seeing a primitive form of polymorphism. Effectively you have what looks like one procedure (or in the case of classes, method) that operates differently depending on what parameter type it receives. Polymorphism in all its polymorphic glory is beyond the scope of this article and isn't critical to a basic understanding of Clarion OOP and ABC.

Composition

The fourth volume of this trilogy (apologies to Douglas Adams) is composition. Although not included in the classic threesome, composition is actually one of the most important features of OO programming as done by Clarion and many other languages. Composition lets you lump several (or many) classes together into a functional unit.

In my own work I find that composition is responsible for up to 80% of the code reuse in my applications. Object-based languages like VB which don't support inheritance have to rely entirely on composition for code reuse. Fortunately Clarion isn't hamstrung in this way, as AppGen-based development would be impossible or impractical without inheritance.

This handful of concepts and terms comes up repeatedly in any discussion of OOP. As I hope to demonstrate, a basic understanding of each is fairly easy to come by.

In order to understand how OO code works, however, you first need to see how it's structured.

Looking At ABC

ABC generated code is considerably more compact than procedural Clarion code. Here's the code that's used to run a typical ABC browse procedure:

```
    CODE
    GlobalResponse = ThisWindow.Run()
```

Actually there's a little more to it than that. But if you look at your code statement, that's what you get. And on a simple form, you won't even find any source code for a Run() method! Clearly there's quite a lot going on under the hood. Although it isn't apparent from the generated code, the procedure is making heavy use of the ABC library.

As you probably know ABC stands for the Application Builder Classes, and so to understand ABC you need to understand classes.

The Class

The fundamental structure of all object oriented code is the *class*. A class is a sort of group structure which contains procedures as well as data. (In fact, you can use the Clarion deep assignment operator with classes just as you do with groups, though this is not generally recommended.)

In its simplest form, a class is also a bit like a Dynamic Linked Library (DLL). In a DLL, you have procedures, and you also often have some data that is shared by procedures inside the DLL. Similarly, a class typically contains procedures (usually called methods), and data. A class is much more flexible than a DLL, however, largely because of the mechanism of inheritance.

You will sometimes hear the terms object and class used interchangeably. To be accurate, the class is the declaration, and the object is an instance of the declaration. The distinction is important since you declare or define any class only once, but you can have multiple instances of the class in use. For example there is a popup class in ABC that handles browse context menus. There is only one definition of this class, but every browse you create will have its own instance of the popup class. It's not absolutely critical to differentiate between the two, however. Many developers use the term class when they really mean object, and occasionally the other way around. A lot depends on how high the level of "OO purity" is at your location. Then again an OO purist wouldn't be using a mixed language like Clarion.

Declaration and Code

Classes have two parts: the declaration, and the code. I always begin creating a class by writing the declaration.

I frequently need a means of adding some simple message logging to my application, for debugging purposes, particularly when writing code for complex processes. I find it easier to read a log of what happened when than to step through the debugger and try to remember afterward what order various pieces of code executed.

Listing 1 shows some source code which uses such a debugging object. In this example the debugging object is called db.

Listing 1. Code to create a trace log.

```
db.Trace('Before handling event ')
db.Trace('----- contents of listq -----------')
LOOP y = 1 to RECORDS(SELF.ListQ)
  GET(SELF.ListQ,y)
  GET(SELF.FieldColorRefQ,1)
  db.Trace(y & ' - markfield ' & SELF.MarkField & ', nfg ' |
   & SELF.FieldColorRefQ.Nfg & ', nbg ' |
   & SELF.FieldColorRefQ.Nbg & ', sfg ' |
   & SELF.FieldColorRefQ.Sfg & ', sbg ' & SELF.FieldColorRefQ.Nbg)
END
db.Trace('-------------------------------')
db.Trace('Keystate is ' & KeyState())
CurrSelection = SELF.ListBoxFEQ{PROPLIST:MouseDownRow}
IF BAND(Keystate(),0100h) then db.Trace('shift key held').
```

The log created by running the code might look like Listing 2.

Listing 2. An example trace log.

```
001 - Before handling event
001 ------------------- contents of listq ----------------------
001 - 1 - markfield 1, nfg 54234, nbg -1, sfg -1, sbg -1
001 - 2 - markfield 0, nfg -1, nbg -1, sfg -1, sbg -1
001 - 3 - markfield 0, nfg -1, nbg -1, sfg -1, sbg -1
001 -------------------------------------------------------------
001 - Keystate is 33024
001 - shift key held
```

In order to accommodate this functionality I need a class that is able to do at least two things: store trace messages, and display those messages on screen on demand (actually there are a lot of other useful features that could be added, but these two will do for a start).

I begin my debug class by creating the following class declaration:

```
DebugClass     CLASS,TYPE,MODULE('DEBUG.CLW')
TraceQ            QUEUE
Text                 STRING(200)
                  END
Trace             PROCEDURE(STRING)
ShowTrace         PROCEDURE
               END
```

Actually, this isn’t going to work as written. Why not?

Anticipating Encapsulation

I’ve created a class structure that contains both the methods, and a queue to hold the data. One method adds the data to the queue, the other method displays the queue. This combination of code and data into a single structure is an example of encapsulation.

But the queue isn’t going to work the way it’s declared, because Clarion doesn’t allow you to have a queue structure inside the class. That in itself isn’t important – it’s just a "feature" of the language – but what is important is the process you go through to get around this problem, because it’s something that comes up time and again in Clarion OO programming, and in the ABC classes.

References

The way to get a queue into a class without declaring it in the class is to use a reference.

A reference is a sophisticated kind of pointer. A simple pointer points to a location in memory where something is located. The problem with pointers is they’re just addresses – you can make a pointer point to anything at all, which can be quite dangerous. If you have what is supposed to be a pointer to a queue, only it points to a window structure, you’re certainly not going to get the queue data you’re expecting. Worse, what happens to your window when you write to what you think is a queue?

A reference can only point to a location that contains something of its own type. For instance, a reference to a string variable is declared like this:

```
StringRef &string
```

The & (ampersand) character indicates that this is a reference, and that

StringRef can be made to point at any string variable. StringRef cannot be made to point at a byte variable, for instance. If you try, you'll get a compiler error.

To assign a reference you do the following

```
StringRef &= MyString
```

where MyString is a string variable. After you've done that assignment, you can treat StringRef exactly the same as MyString. StringRef is like an alias to MyString. Be sure to type

```
&=
```

and not just

```
=
```

or the reference assignment will not happen!

As I mentioned, you can't have a queue inside a class. You have to declare the queue outside the class, and then create a reference to the queue inside the class. Declare the queue outside the class, as in Listing 3. Then create a reference inside the class of the same type as the class.

Listing 3. The class (and queue) declaration.

```
TraceQueue     QUEUE
Text              STRING(200)
               END



DebugClass     CLASS,TYPE,MODULE('DEBUG.CLW')
TraceQ            &TraceQueue
ShowTrace         PROCEDURE
Trace             PROCEDURE(STRING Text)
               END
```

Note the queue declaration. The reference type is the same as the label of the queue (TraceQueue), but with a & prepended.

If you've worked with OOP you might think Listing 3 is only a partial solution, and you'd be right. There's yet another wrinkle to this whole business of queues in classes, and you'll see shortly why it's important, and why it's a good example of one of the key aspects of OOP.

First, however, you'll need to write the Trace and ShowTrace methods.

Where Do I Put This Stuff?

One of the differences between procedural code and object-oriented code is that a block of procedural code that does one thing is normally contained almost entirely within a single source file. Classes, on the other hand, are typically contained in two source files.

Listing 3 shows the class declaration. By convention class declarations are kept in files ending in .inc, and you can create this one in a file called debug.inc. If you want this class to be available to all your applications, place it in the libsrc directory; otherwise put it in your current working directory. I usually put all generic classes (those which are not tied to a particular class) in the libsrc directory.

Create the class source in a separate file, called debug.clw. This file will contain the source code for the methods (procedures) which belong to the class.

This approach isn't actually all that different from procedural Clarion. In procedural code almost everything is contained in the procedure's module, but somewhere in your application a prototype for the procedure also has to be declared. The main difference is that in legacy applications procedure declarations are generated right into the main source file instead of into a separate INC file which is then INCLUDEd into the source (the ABC templates do take the INC approach with application procedures as well as class declarations).

Strictly speaking you don't have to have your class declaration and implementation in separate files, but it's generally a good idea. One reason is that by keeping your declaration separate from your implementation you can, if you wish, put the actual code in a DLL. Anyone who has the declaration can then use your classes, but they won't be able to see your source.

Listing 4 shows the debug class source.

Listing 4. The class source.

```
   MEMBER

   MAP
   END

   INCLUDE('DEBUG.INC')

DebugClass.ShowTrace         PROCEDURE

window WINDOW('Debug Messages'),AT(,,493,274),FONT('MS Sans
Serif',8,,),SYSTEM,GRAY,DOUBLE
       LIST,AT(5,5,483,246),USE(?List1),HVSCROLL,FONT('Courier
New',8,,FONT:regular),FROM(self.TraceQ)
       BUTTON('Close'),AT(230,255,,14),USE(?Close)
     END

   CODE
   OPEN(WINDOW)
   ACCEPT
      IF FIELD() = ?Close AND EVENT() = EVENT:Accepted
         BREAK
      END
   END

DebugClass.Trace             PROCEDURE(STRING Text)
   CODE
   SELF.TraceQ.Text = Text
   ADD(SELF.TraceQ)
```

All class methods are declared in the form `classname.methodname`, which is so that compiler knows to which class the method belongs. The two methods shown in Listing 4 are straightforward. The Trace method ads a record to the queue, and the ViewTrace method displays a window with a listbox. That listbox has as its FROM attribute the queue:

```
FROM(self.TraceQ)
```

Notice the use of the keyword SELF to refer to the current class. Any variable that belongs to the class (also known as a property) and any method that belongs to the class, must be written as

```
SELF.variable
```

or

```
SELF.methodname
```

Aside from making it easy to refer to the current object, SELF allows the compiler to differentiate between class data and global and method local data.

Testing the Trace Class

Create a small test application, or use one you have lying around. You can very quickly create an application by using Quick Start. This process is described in the Quick Start tutorial in the Getting Started booklet that comes with Clarion.

To test the trace class you need to let the application know about the declaration, and you have to provide the method code, either in source or compiled form. There are several possible approaches, and this is one of them:

1. Include the source file in the project. It's possible to have class source automatically compiled using the LINK directive, but in this case you'll need to choose Project|Edit, and in the External files list add debug.clw. If debug.clw is in the libsrc directory or the current working directory you won't need to supply the full pathname – just the file name will do.

2. Go to the application's global embed list and put the following code in a source embed in the After Global Includes embed point:

```
include('debug.inc')
```

Don't close that embed point yet! There's one other thing that you need to do here. Look back at the declaration and you'll see that the trace class has a TYPE attribute. That means that you can't use it directly. The compiler won't allocate any memory for any class (or queue, or any other data type, or even a method) which has a TYPE attribute. So why am I using this attribute?

I don't have to. But I always generally declare my classes with the TYPE attribute because it forces me to create a specific instance of a class. if I attempt to do this somewhere in my code:

```
DebugClass.Write('test')
```

I'll get the "Cannot use TYPEd structure in this way" error.

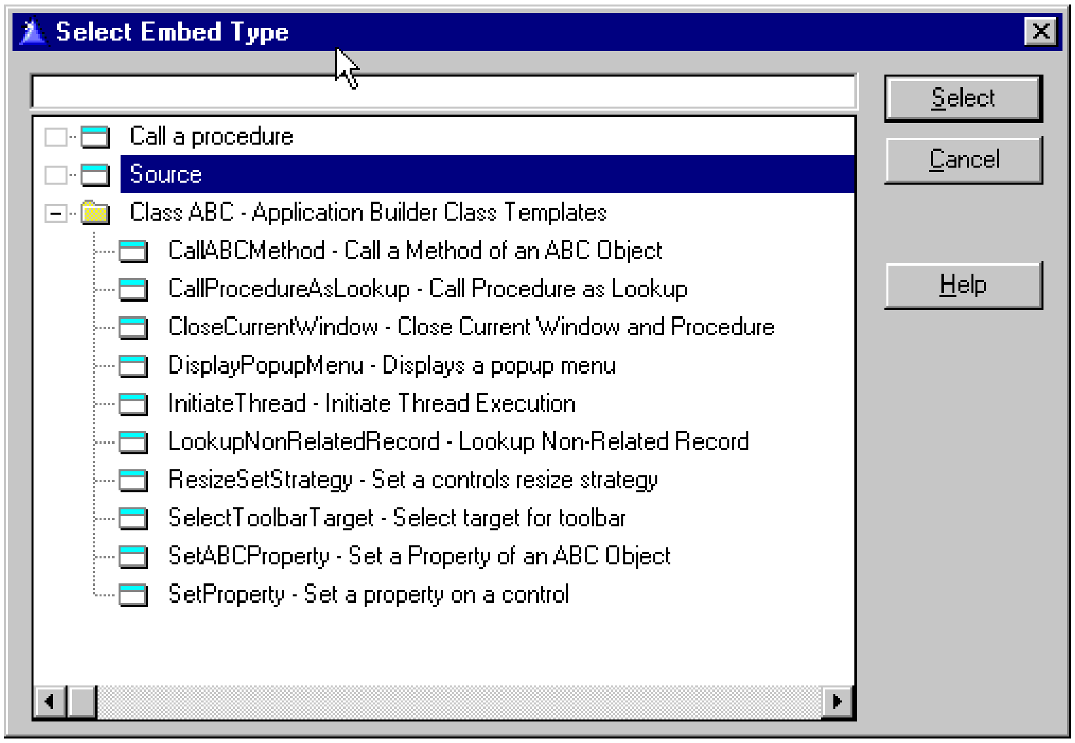
The easiest way to declare an instance of a class is as follows (you can type this just below the `include('debug.inc')` statement):

```
db DebugClass
```

Now you're ready to test the class. Assuming your test application has a browse procedure, go to that procedure embed list. You may find the following instructions easiest to follow if you first choose View|Contract All from the menu.

Under local objects choose `ThisWindow` and expand its embed list. Under `WindowManager` locate the `TakeEvent` method. Expand the embed list and double-click on the `CODE` entry. The Select Embed Type window appears, as shown in Figure 1.

Figure 1. The Select Embed Type window.



Call a procedure
Source
Class ABC - Application Builder Class Templates
CallABCMethod - Call a Method of an ABC Object
CallProcedureAsLookup - Call Procedure as Lookup
CloseCurrentWindow - Close Current Window and Procedure
DisplayPopupMenu - Displays a popup menu
InitiateThread - Initiate Thread Execution
LookupNonRelatedRecord - Lookup Non-Related Record
ResizeSetStrategy - Set a controls resize strategy
SelectToolbarTarget - Select target for toolbar
SetABCProperty - Set a Property of an ABC Object
SetProperty - Set a property on a control

Choose Source, and in the resulting source embed type

```
db.Trace('Event ' & EVENT())
```

Save your changes. Go to the main menu and add a new menu item. Call it Show Trace and on the Actions tab do NOT enter a procedure name. Instead click on Embeds, and on the Accepted event generated code embed point add a source embed with the following code:

```
db.ShowTrace()
```

Save your changes and run the application. If you've followed instructions very carefully, as soon as you either run the browse or click on Show Trace your application will...blow up.

What's This GPF Doing Here?

I'm not in the habit of publishing code that GPFs, and the only reason I'm doing it now is because this is going to happen to you sooner or later, so you might as well get it out of the way.

The problem is that the queue reference (`TraceQ`) hasn't been initialized to anything. It has a value of `NULL`, and if you attempt to use a `NULL` reference, Windows will report a GPF. So you have to initialize the reference.

Constructors and Destructors

Many, if not most, classes have to do some kind of data initialization or object creation. The proper way to handle this is to set aside a class method to take care of this, and often another method to take care of cleaning up the class after you're done with it.

These methods are called constructors and destructors, respectively. And there are two schools of thought on how these methods should be handled. One school of thought (let's call this the DAB school) says these methods should be like any other methods, and you the intelligent and methodical programmer will call them as needed. Typically constructors have a name like `Init()` and destructors have a name like `Kill()`.

The other school of thought (call this the great unwashed who are not of the DAB school, school) says constructors and destructors should be called automatically when the object is created or destroyed.

You can do either in Clarion, because DAB buckled under the pressure and added automatic constructors. For now, go automatic.

The Construct and Destruct Methods

The constructor's job is to initialize the queue reference so the trace() method will have a real queue to work with. Remember that when you assign a reference to an object, you treat the reference the same way you would the actual object.

```
DebugClass.Construct         PROCEDURE
   CODE
   SELF.TraceQ &= TraceQueue
```

When you use a constructor you almost always want to use a destructor as well. In this case all the destructor needs to do is free the queue.

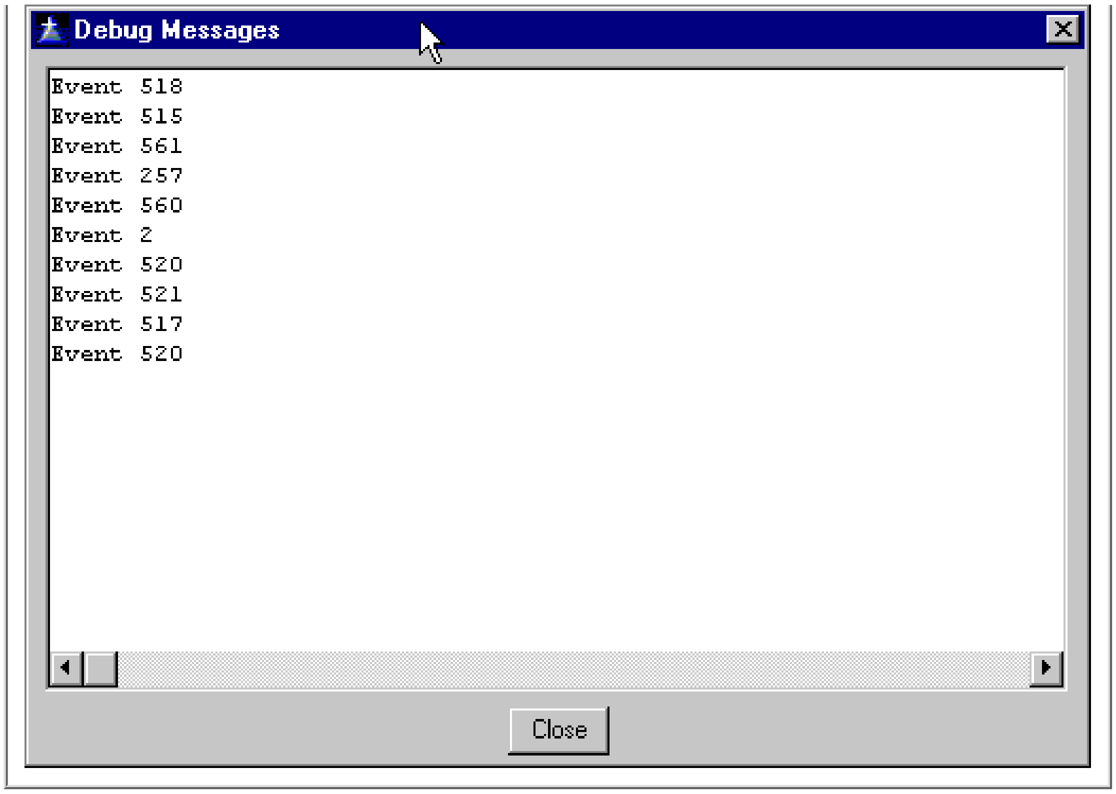
```
DebugClass.Destruct          PROCEDURE
   CODE
   FREE(SELF.TraceQ)
```

Add these two methods to the debug.clw source file. You'll also need to add the method prototypes inside the class definition in debug.inc:

```
Construct          PROCEDURE
Destruct           PROCEDURE
```

Save your changes and compile and run the application. Open the browse and then from the main menu choose the Show Trace option you added. A window similar to the one shown in Figure 2 appears.

Figure 2. The trace window displayed by ShowTrace().



```
Event 518
Event 515
Event 561
Event 257
Event 560
Event 2
Event 520
Event 521
Event 517
Event 520
```

That's how easy it is to use a custom class in your application!

Achieving Encapsulation

There's still one significant problem with this class as declared.

The queue isn't actually part of the class, but is declared outside the class. Whenever possible, you want your classes to be completely self-contained. That's encapsulation – code and data together in a single object.

Since there's only one debug object the current situation isn't necessarily a problem. But what happens if you have two debug objects?

```
dbg1            DebugClass
dbg2            DebugClass
```

Both instances of DebugClass will use the same queue, and that's not normally a good idea. You want any object in your system to be as self-contained as possible. The way to do this is to put the type attribute on the queue, and declare an instance of it inside the class, using the NEW operator.

```
DebugClass.Construct         PROCEDURE
   CODE
   SELF.TraceQ &= NEW(TraceQueue)
```

Since the destructor's job is to clean up anything which the constructor (or any other method) may have created (read allocated memory for), you should now use it to dispose of the queue when the class is finished. If you don't DISPOSE() what you NEW() you'll end up with a memory leak.

```
DebugClass.Destruct          PROCEDURE
   CODE
   FREE(SELF.TraceQ)
   DISPOSE(SELF.TraceQ)
```

NEW is, well, new to most Clarion programmers. In procedural Clarion, the runtime system takes care of memory allocation automagically. The only time you actually allocate or deallocate memory is when you add records to, or delete records from, a queue. In the world of OOP, however, objects are being created and destroyed all of the time. It’s possible you may never do this explicitly in your ABC-related code, but ABC does it lots, and it’s part of the power of OOP.

As TraceQ demonstrates, references have a dual purpose. They can point to an existing object which is declared elsewhere, in which case they function like an alias, or they can be used to hold an object just created, in which case they may be the only reference to the object.

Just as the debug class created the queue on the fly, you can also create the debug object on the fly, if you wish. Instead of declaring your debug object like this

```
db DebugClass
```

declare it like this

```
db &DebugClass
```

By using the ampersand you’re specifying that this is a reference rather than an instantiated object. Before you can use the object in your code, you’ll need to create an instance like this:

```
db &= NEW(DebugClass)
```

and when you’re done with the object you need to free up the memory using DISPOSE:

```
DISPOSE(db)
```

You should always DISPOSE anything you NEW or you’ll end up with a memory leak.

So what’s the difference? In this case, there is no practical difference. But there may be times in your programming when you want to create objects on the fly (you can have a queue of objects, for instance). And sometimes you just want explicit control over when an object is created and when it is destroyed.

The ABC Angle

The ability to create and dispose of objects, and assign references to those objects, is one of the keys to the ABC class library. ABC is continually creating and disposing objects, and working with object references.

Now as useful as all of this is, if you didn’t have anything more, well, you’d really just have something like Visual Basic. And there’s a lot more to Clarion and ABC. In the next article in this series I’ll look at inheritance, another of the keys to understanding the ABCs of Clarion OOP.

> *Note: The debugging class used in this article is a rudimentary form of the cciDebugClass which is part of the Clarion Open Source Project and available by download.*
>
> *An example application with a working DebugClass is also available for download.*

David Harms is an independent software developer and the co-author with Ross Santos of

Developing Clarion for Windows Applications*, published by SAMS (1995). His company, CoveComm Inc, publishes Clarion Magazine.*

Vol 1, No 2
March 1999

CoveComm Inc.

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ

Feature Article

Constant Class Design Insights

by David Bayliss

Behold, for peace I had great bitterness: but thou hast in love to my soul delivered it from the pit of corruption: for thou hast cast all my sins behind thy back.

One of the interesting differences between different people is their perception of events past. Two individuals can experience exactly the same thing, one remember all that is good, the other all that is bad. This difference is most marked when reviewing our own performance. The most interesting point is that we all tend to assume we have the right to our own unique slant on history. Yet we all have a phrase that sometimes troubles our sleep, the "skeleton in the closet".

I don't know where the expression came from but I actually did attend one of the first "schools for poor boys" in the UK (est. 1640) where they did have a skeleton in the closet. A boy who had been punished for cheating by having a lead pencil shoved up his nose. When he died of internal bleeding the headmaster tried to hide his deed by incarcerating the boy in a closet. He was discovered some years later. One of the more sobering messages of the bible is that one day those closets will be opened.

The issue facing me is rather less drastic but still problematic. I must confess that in writing this article I am not quite sure of the correct tack to take. Should I explain the design insights I *should* have had (as I can now see with perfect 20-20 hindsight) or should I detail the evolution of the Constant Class from the germ of an idea into one of the cornerstones of the system?

Given the Constant Class didn't appear in its current form until C5 (and the gaffe is therefore blown) it is probably instructive to take the latter approach. Therefore you will see how a particular need developed into a set of (widely dispersed) code. How this code was then extracted (and abstracted) to form a class that is now generally applicable.

I would also like to acknowledge that although the design of the class was mine the implementation was done by Roy Hawkes of the TopSpeed Development Center.

The Germ

The real germ of the constant class is actually the data set used by the ErrorClass. However to understand the full usage of the Constant Class it is necessary to look at the problem in a wider context.

Just about any program can be broken down into two distinct parts, the algorithm and the data it works upon. Now as database programmers you will automatically think of data-files when I use the term data. But in fact, commonly, the data in the data-files will not be provided by you the programmer, it will be provided by the end-user. (Of course the end-user may also be you!) The data in your actual program will usually be far more diffuse, possibly even hidden, but still vital.

One piece of data you may not think of is the Windows and Reports in your program. They are just data-slabs (resources actually). But there are also many constants "hard wired" into your programs, usually in the form of parameters. Take the following:

```
?MyString{PROP:Text} = ‘The meaning of life the universe and everything is...’
```

That code obviously contains a string constant, right? Actually, no. It contains two string constants and two numeric constants. (`Prop:Text` is a string, `?MyString` is a number as is the implicit `Window $` that precedes it.)

Now at the top of a procedure with a moderately flexible display there will often be ten or more of these lines. In some common totally dynamic cases it is easy to end up with a hundred of these statements (consider setting the color, position, tooltip and icon of a set of buttons on a user-configurable toolbar). This is perfectly reasonable and easy to code but it does have some significant drawbacks:

- Code bloat. Each of the above statements compiles to about 50 bytes. 100 lines is 5K ...
- Inflexibility. Suppose there are two types of text string, for different languages perhaps. That means two sets of code (or `CHOOSE` statements).
- Alterations have to be done in code. To appreciate this one you need to subscribe to the Bayliss Mantra that says "new code is bad code." If you allow that >75% of coding effort happens during the maintenance phase then new code is <25% of the way there. Therefore you should always aim to let your source files get at old as possible. The older the date on the file the more likely it is to work. But if constants are in code then it is impossible to change the occasional misspelling without soiling the source code.
- Detail and complexity are interwoven. Complexity in programming is the sums and the clever bits (such as control positioning). It has to be done very slowly (or very often!). Detail is all the grungy stuff that has to be done well and thoroughly but can still be done quickly. Typically there will be 10x as much detail as complexity, but the complexity takes 10x longer per line to write. If you mix detail and complexity you can end up with 10x as much code that has to be written slowly. So you want detailed constants (such as prompts / messages) out of the main body of your code.
- The "prompt" writer has to program. This is really just another facet of the above two but still important. In the ABC system I want the prompts to be alterable by people who may not be happy trying to penetrate my coding. Therefore I need to be able to identify the *safe zone* (the .trns) from the *DAB zone*.
- Lines full of constants can weigh heavily on the 16 bit compiler’s internal data pools leading to the dreaded dynamic pool errors.

The First Solution

With the above somewhat forbidding list weighing on my mind I retired to slumber and awoke in the early hours with a partial solution. Why not use a statically initialised group to act as a data stream that can be parsed apart and then used. The group can reside anywhere and you have a separate piece of complexity to parse the data structure. One key extra thought is that you can use pstrings or cstrings in the data structure. This means there is no wasted space. Fairly rapidly I wrote the code for the error class (see `ErrorClass.AddErrors`, in aberror.clw). After about 20 lines I was home.

Have you ever noticed that when you get a new tool for your toolbox there are a whole host of jobs that arise to which the tool is perfectly suited? Within a few weeks we had five other similar pieces of code scattered around the ABC system. Then the synchroniser team spotted the trick and did the same. Come the spring of ’98 the Wizatron team had also rolled out five or six copies of similar code.

Now you might think, hey, why not use procedures, then you only get one copy of the code! But the task is not that simple. The thing is that although each of these pieces of code used the same idea, they each did different things with the data, and their data was a different structure. For example, in abtoolba.trn the structure is simply ushorts followed by strings. The print-preview has two strings and a short. Sometimes the data was being used to pre-fill a queue, sometimes in property statements of a window. Although the germ was the same between each block of code, very little of the code was actually the same. Also each lump of code was quite easy to write so the easiest way for people to reuse the code was simply to copy it from somewhere else and hack it a little bit.

OK is not OK

This is the easiest approach, but not the cleanest. The problem is that the low-level of the constant class code (as it then wasn't) is actually extremely dirty. It makes assumptions about the layout of Clarion groups & the data structure of clarion data types. The thought of having >20 copies of that code floating around the system was not pleasant. Also the Wizatron team needed the code to go as fast as possible, that means spending time tweaking; we didn't want to do that many times. It came down to an engineering decision. Do you put up with a sub-optimal solution in order to make life easy or do you hammer at the problem until you get it right? There is a phrase I always bring up when interviewing people to see how they react: "Ok is not OK."

So at this point we sat down and thought, what actually *can* we abstract from this problem to enable us to write a class. We came up with the following: "The class should have the ability to parse apart an unknown but externally specified data structure into some suitable location or locations(s)".

That wasn't much to go on, but it was enough. We could now see how the class would be used. First there would be an init sequence specifying where the group was, what fields should be expected and in what order, and where the result for that field should go. Then there would be a next loop somehow reading the group record by record. Then some kind of closedown. One could still argue the spec is a little vague, and Roy did look extremely dis-chuffed when I told him my idea, but the result actually works very cleanly.

I will now go through the methods themselves in the rough sequence they would be used. I shall assume you have the actual code to look at (you can find it in abutil.clw).

The Implementation

For the following I will describe the group as the data structure and those fields pertaining to one "set" of data as a record. So supposing I had a group which consisted of 20 pairs of strings. Each string pair I will refer to as a record.

```
ConstantClass.Init
```

Internally the list of fields (that is to say the data types expected and where they should go) is to be stored in a queue, so the `Init` method constructs this. The parameter it takes gives the first level of flexibility built into the system. How do you decide when you have read all the records? You have four alternatives :

. Term:EndGroup - Stop when you come to the end of the group. This is the cleanest and least accident prone (you can't run off the end of the list or *loose* items on the end). The main down-side is it doesn't allow you to arrange for only the first n items of a group to be used.

b. Term:Ushort - The first field of the group is unique (a ushort) which denotes the number of records (groups of fields) to follow. This is efficient and clean but suffers from two types of accident. If the number is too high you get a gpf as the parser trips off the end of the group. If the number is too low the last elements are silently ignored. On the plus side you can select to use only the first n elements.

c. Term:Byte – The same as (b) but limited to 255 records

d. Term:FieldValue - This is the old *magic number* trick. If you specify this then the parser keeps going until it finds a first field of a record that matches the termination sequence (which you have to put into the TerminatorValue field). This is a good lazy/safe solution if you are reading a sequence of strings. Just make the last value '**STOP**' and make that the terminator value. The only real down-side is a slight danger of picking a terminator that someone uses as a data value. The biggest perceived downside is a guilty twinge that proper programmers don't use magic values.

Note that part of the Init function is 'split out' into Reset to allow a group to be re-parsed without going through the whole set-up rigmarole again.

```
ConstantClass.AddItem PROCEDURE(BYTE ItemType,*? Dest)
```

This tiny little procedure is the hub of the class. It is called once for each field in the record. The sequence these are called does matter. It is assumed the first call to AddItem is the first field in the record. The *? Dest is a *field pointer* to the place you want the result of the field to go. The type of the variable passed in does not have to be the same as specified by ItemType. ItemType pertains to the type of the field in the group structure. (That said, using pstrings to store byte values would be a bit daft)

Note the CLEAR is important as the queue contains ANY variables. (If that sentence didn't make sense consult the LRM, the usage of ANY variables in queues is very delicate, but powerful).

```
ConstantClass.Set PROCEDURE(*STRING Src)
```

This is the procedure that hands the group over to the constant class and primes the constant class to read it. Note that you may do multiple sets, so one constant class can process many different groups provided each group has the same structure. Note too that the constant class copies the group so groups passed into the constant class do not need to be static. (This illustrates an interesting trade-off. The constant class is general purpose and therefore is coded to be much safer than the ErrorClass equivalent. The down side is that it is slightly more expensive in use).

```
ConstantClass.Next PROCEDURE
```

You may think of Next very much like the file driver equivalent: read in the next set of data. The only difference is that the data goes into those places defined by the AddItem, not into some separate record buffer. (Re-read those two sentences again 10-15 times. There may well come a time when the concept is important).

The coding is rendered more complex by the need to check for four different methods of termination. Note that upon termination Level:Notify is returned so a simple LOOP WHILE ~MyConst.Next() can be coded.

Leaving aside the CASE SELF.Termination (which is detail, not complexity) the real code starts with the GET of SELF.Descriptor. That is moving the queue so that the results of the first AddItem are available. Now dependant upon the type of the data field the suitable byte-shuffling is done to read out a value which is then placed into the required location. Then the next item in the queue is read to process the next field within the record. It is possible to get confused here. The LOOP inside next is looping through the fields of the record, not the records of the data structure.

Note too that the detail of the exact low-level format is removed from this complex procedure. This procedure is the controller procedure for making sure the right fields are read in the right sequence and sent to the right place. That is enough for one procedure to worry about.

```
ConstantClass.Reset PROCEDURE
```

This procedure is simply to allow the same group to be re-read without an intervening SET. It also handles the reading of the leading byte/ushort (if appropriate) allowing the Next procedure to stay clean (because there is no special case of *first record*).

```
ConstantClass.Kill PROCEDURE
```

This is the traditional way of closing off a class. Mainly there to dispose of the queue and the copy of the group that was taken. Note though the special handling required of the queue because it contains an ANY variable.

```
ConstantClass.Next PROCEDURE(QUEUE Q)
ConstantClass.Next PROCEDURE(FILE F)
```

The preceding procedures are the *real* interface. There are then a couple of procedures just to make life easier. One thing we noted was that very commonly all the fields of the AddItem were actually fields of a queue, and straight after the Next we would do an Add. So we made that case easier. You still pass in the fields of the queue but rather than code the loop yourself you pass in the queue and the work is done for you. This is done similarly in the file case.

```
ConstantClass.GetByte PROCEDURE()
```

Now, in all the above one vital little detail has been deliberately left out. How do you parse the group structure to give all this high level information? There are essentially two different problems, the first is keeping your place in the byte stream; the second is plucking the various data types out of this byte stream.

In typical fashion these two problems have been teased apart. GetByte tackles the first. After a Reset the first GetByte will return the first byte of data from the record stream *after any leading byte or ushort (to give the record count) has been read*. The next call to GetByte will get the second byte, etc. GetByte has dual usage. It is used by all the other Get functions as a primitive but can also be used in its own right to read in fields of type byte.

```
ConstantClass.GetUShort PROCEDURE()
```

Ushorts are stored with the least significant byte first so this is read (using GetByte), then the second byte is read, shifted left 8 and added to the first. You may wonder why we've used Rval. This is because the evaluation order of the operands of the addition operator is not specified so, as the GetByte calls are sequence specific, you need to split the expression to define which one comes first. In other words if you have the code GetByte()+BSHIFT(GetByte(),8) it is not defined whether the first or second GetByte call will happen first so the data could end up reversed.

```
ConstantClass.GetShort PROCEDURE()
```

This is precisely the kind of low level dirt that persuaded me this code had to be in a class. Constructing two bytes into a ushort then claiming it happens to work if you just assign that to a short is extremely ugly. But it works beautifully (for now). By encapsulating this code I can ensure it works in the future too.

```
ConstantClass.GetLong PROCEDURE()
```

That natural way to do this is with four `GetBytes` where you shift 8 bits for each iteration of the loop. It took me a while and quite a lot of squared paper to persuade Roy that this trick would work, but it does!

```
ConstantClass.GetPString PROCEDURE()
```

Twigging that constant pstrings could be parsed without wasting space was one of the key concepts behind this technology. It is actually very simple. You just read the leading byte (the length byte) and then you know how many more characters there are to read. Now here again we have traded some *speed* for some *clean*. It so happens that because of the way we store the group (in a copied string) we could code the pstring reader to be quicker by using a string slice and updating the pointer in one go. Instead we have chosen to persist with the `GetByte` interface reading the pstring one character at a time so that we could change the data storage mechanism with a minimal re-write.

```
ConstantClass.GetCString PROCEDURE()
```

Reading the cstring is less complex than the pstring and yields itself naturally to the `GetByte` approach, you simply sit in a loop reading characters until you hit the null terminator.

Conclusion

There you have the evolution of the constant class. I hope the insights into our development process have been of some help, so too the insights into the class itself. For people who like something concrete on which to hang their ideas the simplest example of using the constant class is probably the `SetTips` procedure in abtoolba.clw. `InitializeColors` in abeip.clw is fairly simple too. Using the class to fill a queue in one shot is shown in `TranslatorClass.Init`.

David Bayliss is a Software Development Manager for Topspeed Corporation. He is also Topspeed's compiler writer and the chief architect of the Application Builder Classes.

The Developers' Open Source License

THIRD DRAFT

Version 0.3, March, 1999

PREAMBLE

This license is designed to protect the rights of software developers who wish to release software libraries and software development tools under an open source arrangement.

Most open source licenses are either written with complete software packages in mind or are intended to cover code which is created using open source software tools. This license is specifically designed to accommodate developers who use open source software in the context of proprietary software development, or who work with proprietary software development tools.

TERMS AND CONDITIONS

1. Definitions

The term "source code" is used in this license to mean any program source code, template, project file, or other human readable source file which is used in the creation of software programs.

The term "original author" is used in this license to mean the person who creates the first version of any work placed under this agreement.

2. Placing software under this license

Source code may be placed under this license only by the original author, and only when the original author has the full legal right to the source code. All source code placed under this license must bear a copyright notice by the original author as well as a notice of this license agreement, as specified below. Such source code is referred to in this agreement as "covered source code."

3. Derived works

With the exception of certain kinds of object-oriented source code, as noted below, source code that has been derived from other source code cannot be placed under this license. In other words, you cannot obtain freeware or other source code and place it under this agreement. You can only place your own creation under this license.

Object-oriented code is a special case, since you can create a class that is derived from another class which may be available in binary form. You may place a class that is derived from another class under this license EXCEPT when the terms of this license are in any way in conflict with the terms of the license of the parent class. The class from which your class is derived does not automatically become part of this license.

4. Modifications to the software

Anyone who makes modifications to covered source code creates, by definition, a derived version of the covered source code. Derived versions are automatically covered by the same version of this agreement as the version from which they are derived.

All modifications to the covered source code must be commented with a description of the modifications and the modifier's name (and preferably an email address).

Source code which uses covered source code does not, of itself, constitute a modification or derived version of the covered source code. The differentiating feature is the functionality of the source code and the covered source code. Where the source code simply makes use of, rather than modifies, the functionality of the covered source code, it does not constitute a modification or derived version.

5. Upgrading the license agreement

The original author of the covered source code may upgrade this license or replace it at any time, but only by releasing a newer version of the covered source code under the modified or different license.

6. Distribution of source code and binaries.

You may distribute covered source code as you wish, provided it is always accompanied by this license, and the covered source code continues to comply with the terms of this license. You may charge a nominal fee for the distribution of the covered source code if you wish, and any fee you like for support of the covered source code. Binary compilations of the covered source code, except when distributed as part of a compiled program as noted below, must be accompanied by the covered source code and a copy of this license.

If you distribute a compiled program which incorporates covered source code, you must provide an easily accessible notice that the program contains covered source code. If you are distributing a Windows program, such notice must appear on the Help|About window. This notice, at a minimum, must contain the following paragraph, modified to include sufficient contact information so that users of the program can reach you without undue difficulty:

> A portion of the source code used to create this program is licensed under the Developers' Open Source License. For more information contact:(insert your contact information here)

If the compiled program is created entirely from covered source code, you must use the following form, which is identical to the preceding paragraph but omits the phrase "A portion of."

> The source code used to create this program is licensed under the Developers' Open Source License. For more information contact:(insert your contact information here)

Additionally you must do one of the following:

. distribute the covered source code along with your program, or

b. provide the covered source code on requrest to legitimate users of the program, the original author, and to any authors who have contributed to the covered source code.

7. No Warranty

This source code is not provided with any warranty. You have the source code, and it's up to you to ensure that it does what you want. You assume the same risk and responsibility for using this source code as you do for source code you create.

END OF TERMS AND CONDITIONS

To apply this license

To place source code under this license simply add the following information in each source code file where possible, or in an accompanying text file if it is technically impossible to place the notice in the source code file. The copyright date and copyright holder are shown as examples and must be changed to reflect the actual license information.

> Copyright (c) 1999 Kewl Software Corporation
>
> This software is licensed under the terms of the Developers' Open Source License. A copy of this license has been included with this source code. The license is also available (as of March 7, 1999) at
> http://www.clarionmag.com/common/dosl.html

Vol 1, No 2
March 1999

Feature Article

The Novice's Corner: Understanding Templates and Embeds

By David Harms

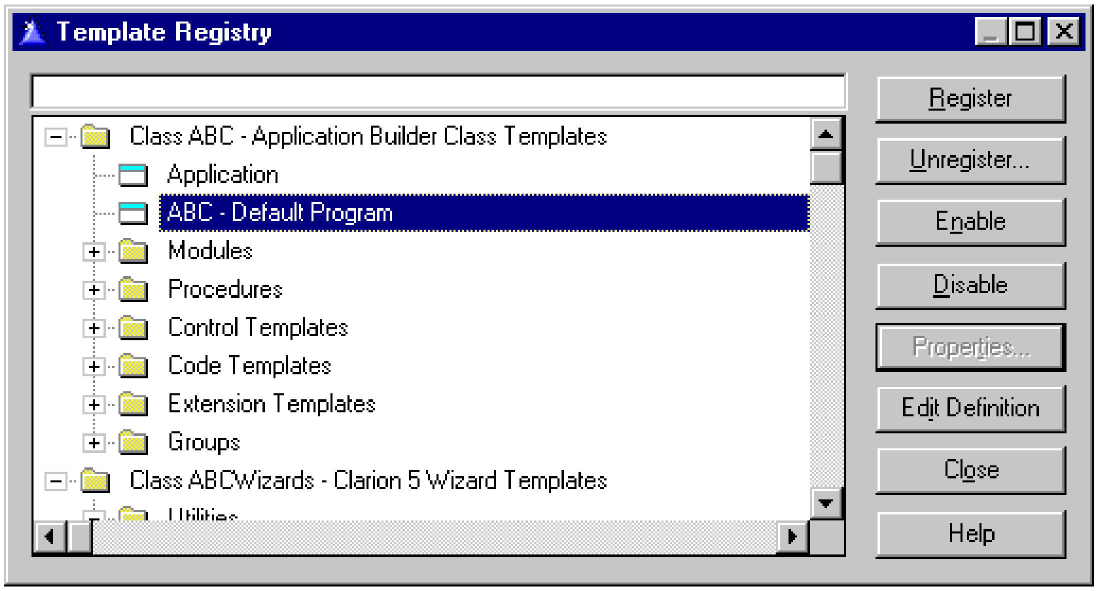
In the first article in this series I examined the some of the basic concepts behind Clarion's application generation technology and reviewed the initial steps most Clarion developers take when creating an application. If you follow the approached discussed in that article you can get an application up and running in short order. Chances are, however, that it won't be quite the application you want, and you'll need to make some significant changes.

If you're working within the Application Generator (AppGen) environment, as almost all Clarion developers do, then much of what you do is predicated on the use of templates, as explained in the previous article. Unlike earlier versions of Clarion, which had a fairly monolithic template structure, Clarion for Windows builds applications out of several different kinds (or classes, though in this case the term has nothing to do with object orientation) of templates.

The Template Registry

You can see which templates are available by closing any open application and choosing Setup|Template Registry from the main menu. Figure 1 shows the Template Registry main window.

Figure 1: The Template Registry, showing the ABC template chain.



Templates are listed in the registry by type. Keep in mind that the templates are simply text files containing template language (and sometimes also Clarion language) statements. You can register and unregister template sets (also called chains) as you wish, although you should keep in mind that not all templates are created equal, and some are less dispensable than others. Most Clarion applications are built on either the ABC or Legacy template chains and many of these apps also use a number of third party templates which provide additional features but generally do not replace the standard Topspeed-provided templates.

Template Files

You'll see two extensions for template files. The Template Registry can load in TPL files, and TPL files may contain INCLUDE statements that point to TPW files, which contain additional template code. For instance, the ABC template chain's TPL file is called ABCHAIN.TPL, and that file INCLUDEs a number of TPW files, all of which begin with AB (and which can be found in the \template directory). If you purchase a third party product which comes with a template (and most do) you'll receive at least one TPL file and perhaps TPW files, depending on how the product author has chosen to write the templates.

Application/Module Templates

The first two templates shown in Figure 1 are the Application and ABC-Default Program templates. These templates control global options for application generation. Similar templates exist in the legacy template chain (CW.TPL). These templates control core AppGen functionality and do it well enough that there are, to my knowledge, no third party replacements available, or really needed (at least not for generating Clarion code). Some developers do make changes to these templates but such customization is less necessary with the ABC templates than with the legacy templates. If you're starting out with Clarion, you can quite safely ignore the Application and Default Program templates.

Below these two templates is a category called Modules. Module templates control the creation of source modules, and there are in ABC five module types: ExternalDLL, ExternalLib, ExternalObj, ExternalSource, and GENERATED. Again, if you're just starting to work with Clarion, you really don't need to know about these. The one you'll use on a regular basis is GENERATED, but as this template will be called as necessary when you create procedures, its operation is almost completely hidden.

All of the other module types allow you to use non-generated procedures which may be in source, lib, obj, or dll form. If you're using external procedures (most likely because your app has grown large enough that it becomes practical to split it up into dlls) then you may well find that you're better off using source includes rather than these templates. But that's a subject for another article.

Procedure Templates

The place you will begin encountering templates is at the procedure level. Any time you create a procedure (except when you do so using an application wizard or a wizatron) you'll be asked what type of procedure you want. The available choices in the ABC template chain are as follows:

Browse	Displays records from a table (file) in a list box and optionally allows the user to add/delete/update the records
External	A procedure that is defined outside the application (this will result in the creation of one of the external modules)

Form	A window that allows for changing a table (file) record. Tabs are automatically used so that large records can still be displayed. If child records exist corresponding browses with update buttons will be added (and update forms for the child browses will be created as needed)
Frame	A main window for an MDI (Multiple Document Interface) application. You need to create your main procedure using this template if you want to use MDI child windows in your application. MDI windows are clipped by the application frame and if moved beyond the frame cause scroll bars to appear on the frame. This is the default for Clarion applications.
Menu	A main window for an SDI (Single Document Interface) application.
Process	A procedure that loops through a specified set of records and (typically) updates those records.
Report	A report with a progress bar and optional previewing.
Source	A procedure for which you must supply all of the code. This is just a framework for the procedure.
Splash	A splash window that appears while the program is loading.
ToDo	A placeholder for a procedure named but not yet defined.
Viewer	View an ASCII file in a list box.
Window	A generic window. You have to populate the controls you want.

The procedure templates you'll most commonly use are Browse, Form, and Report. If you don't use an application wizard or wizatron to create your initial application then you'll also need to create either a Frame (most likely) or a Menu (quite unlikely) procedure as the main menu. You may also want to create a splash procedure.

You may also want to use the Window procedure template. In fact, most of the procedures in any given application (that is, the Browse and Form procedures) are really nothing more than a combination of the Window procedure and one or more control templates.

Control Templates

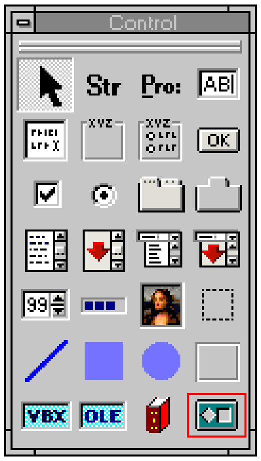
While Clarion for Windows 1.0 was in beta testing the then template set (what is now referred to as the Legacy templates) went through a radical transformation. In the early betas the templates that created a browse or form were monolithic. There was one template for a browse, another for a form, another for a report, and so on. This worked, and was in keeping with the approach of the ancestor to the template, the model file, but it wasn't the most efficient use of code.

Enter control templates. In any procedure a certain amount of code is devoted to managing standard window behaviour, and a certain amount is used to manage windows controls.

A control is anything that appears on the screen, such as a button, or an entry field, or a list box. A control template is a collection of template code and Clarion language code which acts as a wrapper for a control. And by abstracting the control code into a template, Topspeed made it possible to plug and play with a variety of controls on the same window, without the window having to know about these controls ahead of time. Do you want a list box on your update form? No problem? Entry fields on your browse? Go right ahead.

If you want to stick with the standard approach, you can still can recreate any browse or form by starting with a window procedure and populating the appropriate control templates. Simply bring up the procedure properties screen, click on Window, and then choose the Control Template icon from the Control Toolbox (see Figure 2).

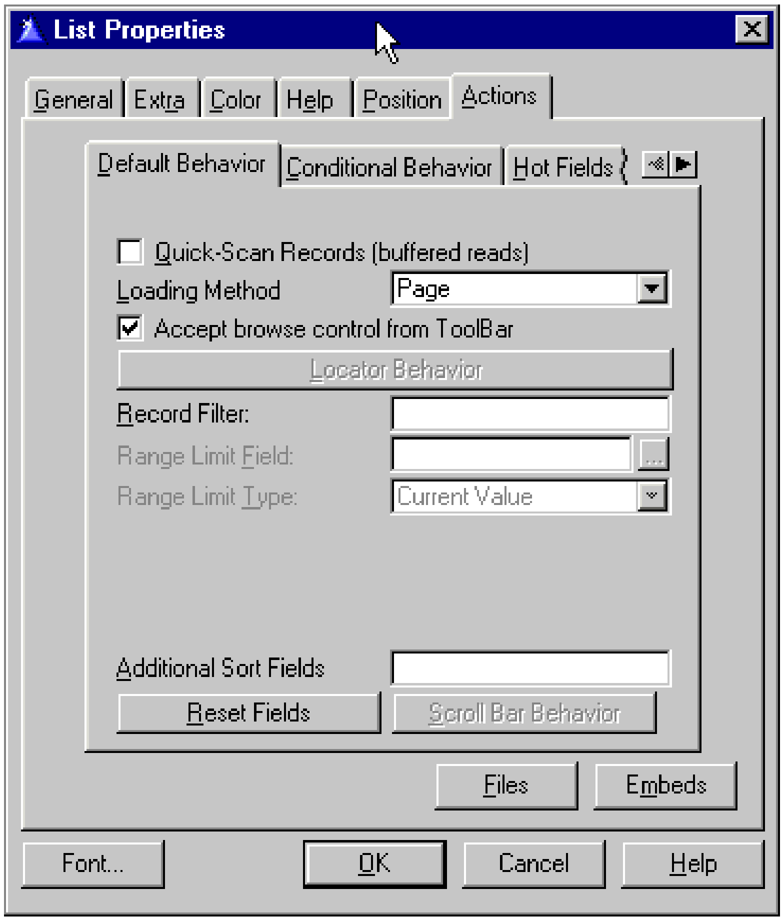
Figure 2: The Control Toolbox with the Control Template icon highlighted



When you populate a simple control onto a window all that happens is the control is added to the window structure. When you populate a control template you're adding much more. Control templates can specify default size and position attributes for the control. They can specify a number of controls to be populated at once. They can require you to enter certain information, such as which field or file the control represents. They can provide prompts for various kinds of optional information. In short, control templates allow you to add some fairly complex functionality to a procedure without having to deal with the code directly.

If the code template has prompts, you can see them by right-clicking on the control and choosing Actions, or by selecting the control and choosing Edit|Actions from the menu. Figure 3 shows the Actions prompts for a standard Browse box.

Figure 3. Browse control template properties



☐ Quick-Scan Records (buffered reads)
Loading Method: Page
☑ Accept browse control from ToolBar
Record Filter:
Range Limit Field:
Range Limit Type: Current Value
Additional Sort Fields

Some control templates are also conditional on other control templates already being populated on the window. You can't put browse update buttons on a window unless there's at least one browse on the window – if there's no browse, the update control template won't even appear in the list.

The ABC templates include control templates for browse boxes, update buttons, query buttons, entry control lookups, file lookups, OLE controls, date/time stamps, and much more. Control templates are also a particularly good vehicle for third party vendors because they can provide standard appearance and behaviour for their controls without requiring the user to write supporting code.

Code Templates

Code templates are usually (though not always) smallish blocks of code that you wouldn't want to have to keep typing over and over again, but which don't necessarily belong to a control or other type of template. Code templates are inserted at embed points, which are discussed below. These templates often have prompts much as control templates do, and the choices you make in the prompts determine how the code is generated.

Extension Templates

Extension templates are a bit of a hybrid of control templates and code templates. If you go to the Extensions list on the procedure properties window, you'll see it includes any control templates which have been populated on the window. Actual extension templates do not have associated controls, however. Often they're used to extend the functionality of an existing control template, or sometimes the procedure template.

Group Templates

Group templates are a somewhat special case, as are the application templates. You don't use them directly as a developer. They're there for the benefit of the template author, and can function either as included template code, or as functions written in the template language.

Templates and Embed Points

Although there's a tremendous amount you can do with templates, they won't always do everything you want them to do. Sometimes (perhaps frequently) you'll want to add your own code to the application.

Clarion is tightly focused on automating code creation. Many of the things you can do with Clarion are possible only because of the work of the templates, but the price is that you can't simply go to the source code and make your changes, because these changes will be lost when you next generate the procedure.

One of the restrictions of the early DOS versions of Clarion was the limited number of places where you could put source code. Most of the logic that determined where source code could be placed was under the control of the Designer application (the forerunner to the Application Generator). In Clarion for Windows, the AppGen has been dumbed down somewhat, if you like, and the templates have been given a far greater level of control over where code can be embedded.

If you click on Embeds on the procedure properties window you'll see a rather lengthy list of embed points. In the ABC templates, most of these will be inside class methods.

For each embed point there is, somewhere in the templates, a template statement that begins with #EMBED. If you want an embed point where none exists (this is not very likely) you can always find the appropriate point in the templates and add one. (You should also inform Topspeed of your requirement as they will most likely want to add the embed point also.)

Each time you use a template, whether it's a procedure template or a control template or any other kind, if that template contains #EMBED statements, those embed points will appear in the embed list.

You can also use the Source view to see embed points in the context of generated code. From AppGen main window right click on the procedure and choose Source, or from inside the embed list click on the Source button. This view of the source code shows you all of the source that is *potentially* generated, rather than just the code that will be generated. As you can see, most embed points cause some standard code to be generated along with your embedded source code.

What About Object-Orientation?

One of the benefits of Clarion's application generation technology is that you don't need to know a whole lot about the underlying language to get things done. The templates provide a simplified interface to the underlying code which, if you're using the ABC templates, is the ABC class library. Of course the more you know about how the underlying code works the more you'll be able to leverage that functionality, but the templates make it possible for someone with minimal or no OO programming experience to effectively use OO code.

Clarion's template technology works equally well with procedural and object-oriented code, although OO code puts less stress on the template language. In a procedural environment the templates have to do much more work generating the core logic, whereas in an OO environment the templates mainly set up calls to the class library, and the core logic is contained in Clarion-only code which makes it a lot easier to maintain.

Templates And Embeds – Finding The Balance

For all Clarion programmers except those who write everything by hand, and those who write nothing by hand, application development is an exercise in balancing templates and embedded source. If you're writing a great deal of embedded source, then there's a good chance that some or much of this code could be better placed in templates.

Templates aren't the answer to everything, of course. There are downsides. When you change a template you have to reload the application, and some template changes force global regeneration of the application source, which can take time, particularly if you're doing a lot of changes. As well, traditional means of code reuse such as class libraries and plain old procedural source libraries may be useful.

Templates become particularly useful in two situations. First, where the code that's to be generated is variable, templates offer tremendous flexibility. The template language is a superset of the Clarion language and has a full range of constructs which you can use to generate code based on prompts answered by the developer.

Templates are also invaluable where you need information that is specific to the application, such as fields in a data file, controls on a window, and so forth. The templates have full access to everything the data dictionary and the application generator know.

If you'd like to learn more about how templates work, I strongly suggest reading through the template language section in the help. There's a vast amount of information, and while it might not all make sense right away it'll give you a good idea of the power of the template system.

Information about specific Clarion templates is also available in the general help. Search for Templates and then click on the Display button to bring up a list of available template help topics.

Next in the Novice's Corner – Building Applications

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, *published by SAMS (1995). His company, CoveComm Inc, publishes Clarion Magazine.*

Vol 1, No 2
March 1999

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ

Feature Article

Topspeed Australian DevCon Reports

Clarion Magazine is pleased to present two special reports on the recent Australian DevCon held in Sydney, NSW, Australia on March 13 and 14, 1999.

If you're attending a Clarion conference and would like to be a reporter for Clarion Magazine, send an email to editor@clarionmag.com.

On a personal note, I'd like to particularly encourage US developers to make the trip down under for a conference. The exchange rate (of about AU$1.50 to US$1.00) means you can probably take a two week trip to Australia for not much more than a two week trip to Hawaii (and maybe less! - it doesn't take long to make back the difference in airfare). I had a terrific time on my tour down under, and a Clarion conference is a perfect excuse for the trip.

Dave Harms, Publisher

Aussie DevCon Report - John Thorley

Aussie DevCon Report - Warren Marshall

Aussie DevCon Report

By John Thorley

This year's Australian Clarion DevCon was the seventh of the 1990's, and was held in Sydney on the week end of 13 –14 March. Numbers were down a little on previous years but there was no shortage of international Topspeed and third party developer support.

Roy Rafalco, Antonio Rajan and Scott Ferrett from Topspeed and third party developers Andy Stapleton, Bruce Johnson, Pete Halsted, Cliff Court and George Willbanks all gave interesting presentations. As well as attending the conference, Andy and Bruce are presenting a number of well-supported training courses while in Australia.

The social highlight of the conference was a BBQ and Jazz Cruise on Sydney's fabulous harbor on the Saturday night. Rhys Daniels, conference organizer, is still claiming credit for the spectacular fireworks display put on for us by the locals.

Highlights

- This conference saw the official launch of Topspeed Consulting Australia. Rhys Daniell, CEO of the Aussie incarnation of Topspeed Consulting was quick to point out that his team of talented developers were keen to take on local and international projects. Rhys noted out that with the Australian dollar so low relative to the US dollar, it made good economic sense to consider outsourcing US development projects to the talented Aussie development team.
- Scott Ferrett advised that a candidate patch for Clarion for Windows release 5A was now available for download and if the patch performed to spec, an official, full CD release would be made available shortly.
- Roy Rafalco announce that the price of Clarion 5 Professional edition would increase on 1st April 1999. Roy also indicated that Topspeed was encountering some corporate resistance to the Enterprise Edition because some corporates perceived the price as too low for them to consider using it for strategic development. Does this mean that we can also expect to see an increase in the price of the Enterprise Edition or perhaps the introduction of a Corporate Bundle? Roy wasn't saying!
- Roy also advised that Topspeed would be withdrawing its presence on CompuServe before the end of April.
- George Willbanks demonstrated a set of Training CD development tools based partly on Lotus ScreenCam and announced that Andy Stapleton had agreed to develop with George two CD-based training courses on generic SQL and SQL/Anywhere. Pricing and availability will be advised in due course.

The award for the best throw away line has to go to Pete Halsted. For those of you who have not met Pete, he's built like the proverbial brick outhouse. Pete had the last presentation before lunch and his presentation was late starting. He reassured us that he'd finish before lunch ... *"Hell, I ain't ever been late for lunch!"*

New Products

Capesoft (Bruce Johnson) released three new products

- Special Agent
  This product integrates a redistributable Microsoft Agent OCX together with templates to enable you to use a talking Teaching agent within your application. You can use the agent to run demos or provide training within your application. While perhaps not to everyone's liking, it's certainly worth a look if you want to impress your users. The DevCon price until the end of March is $AU219. Normal price will be about $AU329.
- Makeover
  This tool comes with 15 styles which you can apply to your application. This can give you features such as wallpaper textures, colored tabs on browses, additional cursors such as a hand instead of an arrow to select buttons. You can add to the style range using the tools provided. Until end of March Makeover is selling for $AU49. Usual price is about $AU79.
- Ticker Tape Control
  This control lets you place a moving sign control on your windows. Special introductory price is $AU9.

Our next Aussie DevCon will be in March 2000 in the Olympic city of Sydney. Why not start planning now to join us in 2000 for a Millenium DevCon to remember. The weather is great this time of year!

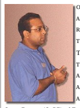
Aussie DevCon Report

By Warren Marshall

Aussie DevCon was held on March 13 and 14 at the remarkable Australian Technology Park at Eveleigh, a few minutes from the centre of Sydney. The venue deserves a line or two here. It houses dozens (hundreds?) of established and fledgling technology companies, as well as a conservatorium of music, in the revamped buildings of what used to be Sydney's huge railway maintenance facility. It's been implemented really well and, in my opinion, the conference success was due in part to this exceptional venue.

Compared with the last Australian conference, this one had a great deal more technical "meat", and also provided much more time for interaction between conference attendees. Lunches were buffet-style, which made it easier to move into and out of various conversations. On the point of attendees, the numbers were dramatically down on last year, possibly due to higher conference pricing. I guess it's a warning to would-be conference holders: if you up the price, you'd better commit resources to retaining that part of your clientele which is price-sensitive. Numbers were, I think, about 60, which is about half what I would have expected, given the outstanding line-up of international and Australian presenters.

Client Day



On the Friday before the conference, Clarion developers were invited to drag clients and would-be clients to see what Clarion had to offer them. They were introduced to Roy Rafalco (TopSpeed's CEO) and Antonio Rajan (TopSpeed's VP Sales), Rhys Daniell of the just-launched TopSpeed Consulting Australia, and Andrew McPherson of Radware, Australia's Topspeed distributor. Marketing demos showed them just how much more effective Clarion was than traditional tools.

Antonio built an Internet connect application from scratch in a couple of minutes and received the appropriate level of amazement from the guests. Juergen (Yogi) Loechner of Qikpay talked about how Clarion saved his marriage and allowed him to capture 60% of the Australian payroll market.

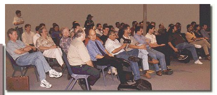
Rohan Dunstan of First Ecom, a large E-Commerce provider based in Adelaide, showed how Clarion Internet Connect applications can be housed at modest cost with First Ecom to reduce the initial financial burden on developers or users.

The numbers at the Client Day were disappointing but those who attended were pretty excited by what they saw. I'm sure we will see a lot of success from this type of format in the future.

The Conference

The first two conference presentations were by Roy Rafalco, CEO TopSpeed Corporation, and Antonio Rajan, VP Sales.

Apart from these two presentations which covered marketing, strategies for the future and the relationship between TopSpeed and its developer base, all the other sessions were technical.



As we have come to expect from previous visits by Antonio and Bruce Barrington (now TopSpeed's Chairman of the Board), Roy and Antonio are very easy to talk to and make excellent ambassadors for TopSpeed. Among the highlights of Antonio's presentation, which mainly centred around the power of Internet Connect and the future music of Wizatrons, was a casual comment that Clarion 6 would be entering beta in April. Antonio then looked at a completely astonished Roy Rafalco at the back of the hall, laughed and said "Gotcha!".

Antonio brought more laughs from the crowd when he tried to convince them that a 60-hour development would reduce to 2 hours with the use of Wizatrons.

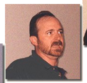
Scott Ferrett (SuRF) of TopSpeed's development team in London gave a warts-and-all demonstration of Wizatron technology (not his area of expertise). It was interesting to see the difference between Wizatrons in their current beta and the picture Roy and Antonio paint – there is obviously a very long way to go yet.

Bruce Johnson of Capesoft (a South African company which produces some of Clarion's most useful accessories) explained the tricks and techniques that convert communication with hardware devices from a mysterious black art into a relatively simple process.

Cliff Court, Managing Director of TopSpeed South Africa, presented a remarkable tool which has just been released. Called Dev Monitor, it tracks and times every activity you undertake in the Clarion IDE and generates costing analysis for your development! As such it may well represent the most significant advance in project cost estimation since the development of computers. Of everything we saw at the conference, this product was one of the crowd-stunners.

On Saturday night, about 60 attendees and partners boarded a small ferry for a delightful harbour cruise around the Sydney foreshores. A band playing inside, plenty of room outside (especially while it rained briefly), an Aussie barbecue on deck, and drinks flowing freely made for a very special night.

I don't know how Rhys and Caroline Daniell organised it, but at 9:15pm Sydney Harbour was lit up with 15 minutes of spectacular fireworks. I assume we will get the same deal at Florida Devcon.



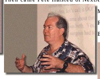
Andy Stapleton of Cowboy Computing Solutions started Day Two's proceedings. He went systematically through the benefits and pitfalls of SQL in Clarion and of specific SQLs. He made it plain that his favourite SQL under Clarion is Sybase SQL Anywhere due to its simplicity, elegance, scalability and the fact that Clarion Enterprise includes an unlimited distribution license. In stark contrast, Oracle was decried as about the most expensive, in hardware, software and administration costs.

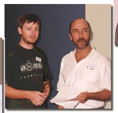
As an aside, it was pointed out that Microsoft SQL and Oracle can not be backed up, except by means of a disk image, since they both use relative disk positioning to quickly locate records.

George Willbanks, president of the Philadelphia NE Clarion Users Group and a long-time Clarion user with a string of corporate presidencies to his name, explained why he regards Pervasive.SQL as the best thing since sliced bread. He went on to detail the steps necessary to implement Pervasive.SQL either on its own or in a hybrid database system with Btrieve.

George then previewed his newly patented CD-ROM based help system, based around Lotus ScreenCam and several other tools. The example he showed was a tutorial on creating a Basic to TopSpeed file conversion program. The technology looks very promising.

Then came Pete Halsted of NextAge Consulting. He showed us the code he used to tame the imaging OCXs which come with Windows. In addition to being enlightening about scanning and image storage, it also raised many issues regarding OCXs within Clarion, including issues of stability, memory leaks and complexity of the interface.

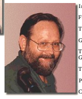




Michael Summons of Summons Technology provided an overview of object-based programming, arguing that this was the appropriate way to write code. This was followed by Jeff Ferguson of University of Western Sydney, who examined the object oriented Systems Analysis and Design paradigms.

Following on from these presentations, Rhys Daniell of Topspeed Consulting Australia presented a counter-view that object-oriented programming was in fact less productive that procedural programming.

Bruce Johnson countered that the issue was not "procedural or object-based" but "Legacy or ABC." About an hour of debate ensued, with no particular winner or loser. Surprisingly (it surprised TopSpeed's people too), on a show of hands, about half the attendees had in fact deployed ABC-based systems and used ABC templates for all their new development.



In amongst the formal proceedings, there were two show-and-tell sessions, where some excellent templates and tools were displayed, all discounted for attendees.

First came the overseas offerings: Bruce Johnson showed some of Capesoft's tools. Cowboy showed his SQL templates. Pete showed his imaging templates. Cliff Court showed his Mail and Fax templates.

Then a couple of Australian tools: Ray Creighton of Sable Software presented his App-Ref, a tool which automatically builds a huge amount of cross-reference information about one or several Clarion applications, allowing you to see unused fields, keys, files, common procedures across multiple apps, etc.

Geoff Bomford of Comformark showed his insanely cheap utility for automatically building complete Windows and HTML Help from your apps.

The conference ended with a short question/answer session with Roy. Not much new knowledge was divulged, but Roy did announce that within a few weeks, TopSpeed's presence on Compuserve would disappear. This would happen as soon as the upload/download infrastructure of GO TOPSPEED was duplicated on the Net.

The conference went extremely smoothly, largely due to the prodigious effort put in by Rhys Daniell.

Product and Service Suppliers at Aussie DevCon 1999:

Capesoft (File Manager, MakeOver, SecWin etc.) – www.capesoft.com
Comformark (Windows Help Builder Templates) - gbomford@acay.com.au
First Ecom (Internet Connect housing and E-Commerce) – www.first-ecom.hm
George Willbanks (Video Training) - www.tsres.com
NextAge Consulting (Imaging Templates) – www.thenextage.com
Radware (Australian TopSpeed distributor) – www.radware.com.au
Sable Software (App-Ref utility) – www.clarion.org.au/app-ref
Stealth Software (Mail and Fax Templates, Dev Monitor) - home.mweb.co.za/jo/joe-vn/index.htm
TopSpeed Consulting Australia (software development services) – www.topspeed.com.au

Other software is available through TopSpeed or its local distributors

Vol 1, No 2
March, 1999

CoveComm Inc

Clarion magazine

Clarion News

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

March 29, 1999

etc 2000 On-line Poll
The 1998 East Tennessee Clarion conference in Gatlinburg was one of the best Clarion conferences anywhere, anywhen, drawing attendees from around the world. Bob Dobbins and Lee White are considering a sequel. Encourage them. Take the survey and tell them what you'd like to see.

Go To Lunch (while your computer compiles)
Steve Parker has a new release of Go To Lunch (GTL), an unattended project compile manager. GTL supports APP and EXE having different names and located in different directories, as well as PRJ files.

Makeover/Tickertape Prices To Rise
Capesoft's introductory pricing on Makeover and Tickertape ends March 31. Makeover in particular has been getting good reviews in the newsgroup.

New Send Internet Mail Release
Princen Information Technology has released a new version of its popular internet mail tool. The company also now has a news server for product support: News.Princen-IT.nl

Split Bar Template In Beta
Quantum Dynamics has a split bar template in beta testing which lets you divide your window into three parts for dragging and resizing (as in Explorer and Outlook).

Pea Brain Software Releases Localizer 1.2000
The newest version of localizer supports currency and date conversions in each language you create.

March 19, 1999

Topspeed To Drop Compuserve Forum
At the Australian DevCon in Sydney Roy Rafalco, Topspeed's President and CEO, announced that the Compuserve forum (GO TOPSPEED) will close in the near future.

Linder Software Announces Install Program Beta
Linder Software will be releasing a beta of version 3 of its SFX Setup-Builder install program on March 19th. The upgrade is free to owners of version 2. Setup-Builder 3 is priced at US$99, or you can buy version 2 for US$69 and get the free upgrade.

Web Server Printing Toolkit Supports Clarion
Automated Solutions Group has released version 2.0 of their Web Server Printing Toolkit (WSPT). WSPT is now available in both OCX and DLL format and includes Clarion header & library files as well as a hand-coded example. Templates are expected soon. WSPT is a server-based (as the name implies) product that redirects printed output directly to PDF files or our own client/side printing format. There are no

3rd party packages to purchase although the company does offer a competitive upgrade to Acrobat Exchange.

March 9, 1999

Technical Site Of the Week

Whether you're looking for a definition of a TLA (three letter acronym) or just some basic technical information, this site has a lot to offer, including alphabetical and topical organization, recommended books, and all the file extensions in the world defined.

StealthSoft Contact/Support Email

Mail and Fax Template and Dev Monitor users please note that StealthSoft has a new contact and support email address: stealthsoft@mweb.co.za

Clarion on IRC

You can join other Clarion developers on IRC most Tuesdays and Thursdays at 24:00 GMT, and Saturdays at 17:00 GMT. That's 7 p.m. and noon EST, respectively.

Looking For Clarion Developers or Clarion Work?

TopSpeed Recruiting and Placement supports the staffing needs of Clarion development groups as well as the career goals of Clarion developers. Visit the placement page at http://www.topspeed.com/placement.htm,

March 1, 1999

Open Linux Project For Topspeed Newsgroup

Clarion Magazine is hosting a newsgroup for the Open Linux Project for Topspeed started by Ron Schofield. This is a private newsgroup and you need to email Ron for the access information. Clarion Magazine subscribers automatically have access to this newsgroup.

EuroDevCon 1999 in Amsterdam

This year's EuroDevCon will be held on April 21 - 22, 1999 at the Golden Tulip Barbizon Hotel in Amsterdam, The Netherlands. The theme of this year’s conference is Run with the Magic. There will be information and demonstrations on the many new features of Clarion 5 including the breakthrough Wizatron technology. Guest speakers will demonstrate the ABC templates and Data Modeller, end user reporting, and intranet/extranet deployment. Speakers and presenters include Roy Rafalco, Ron Tolido (keynote), Bob Foreman, Axel Fenger, Mike Hanson, Mike Pasley, Jim Morgan, Phil Will, Leif Olsen, Richard Chapman, Greg Gubrud, Scott Ferret, Jens-Achim Frei, Erik Pepping, James Fortune, Roelph Du Preez, Randy Goodhew, and Bruce Johnson. For more information visit http://www.itevent.com/eurodevcon.htm.

New Products From Capesoft

Capesoft has released two new products: Makeover and TickerTape. Makeover is a template that dynamically applies styles to your application, based on the number of colors the user's color depth and the procedure type. A style file editor is included. Supports ABC and Legacy. TickerTape is a screen control template for putting a scrolling string on a window.

Read the February News

Do you have a news story or press release we should know about? Send it to editor@clarionmag.com

Vol 1, No 2
March 1999

Clarion magazine

The How and Why of WHO(), WHAT(), and WHERE()

by Gus Creces

A good friend of mine made a fortune in the industrial automation business building advanced "power and free" overhead-conveyors for automobile manufacturers.

For fifty years overhead-conveyors were simple, linear loops, able only to move fenders from stamping to welding, from welding to grinding, from grinding to painting, from painting to assembly and so on at a fixed pace. When faster processes got ahead of slower ones, semi-finished goods were generally labor-intensively stockpiled on the floor and loaded back onto the conveyor as slower processes caught up.

My friend's power and free conveyors are anything but simple or linear. When one process outpaces another, the conveyor shuttles some of the parts its carrying into one of several random-access storage loops until they're needed. These conveyors are essentially modeless. They don't require separate stamping, welding, grinding and painting runs. Fenders, bumpers, hoods, doors and trunk lids in various states of completion are loaded onto the conveyor as required. Parts find their way to the next process at a pace matching that process' throughput. Since parts are uniquely identified and randomly accessible their state of completion determines which process they are shuttled to next, until they eventually make their way to the final assembly line to be incorporated into a vehicle.

The difference between traditional overhead-conveyors and the power and free variety is the level of abstraction incorporated into the conveyor design. By abstraction I mean that the conveyor's design doesn't commit it to carrying a specific part to any predetermined place at a predetermined pace or time.

Abstract Computer Code

There's a strong parallel between these conveyance devices and computer code. The parts handled by an overhead conveyor are paralleled in the data manipulated by a procedure or application. Generally speaking, traditional, procedural code is less abstract than object oriented code. Well-designed OOP code, like its power and free conveyor equivalent, should be able to handle a variety of data more flexibly than its procedural ancestor.

Remember that my friend's more abstract power and free conveyors do not require that all parts to be carried are pre-identified when a conveyor is constructed. The design only demands that the part be identified (read: referenced) when passed to the conveyor. From that point on, the nature and condition of the part (read: its properties) determines its ultimate route through a wide variety of conveyor-accessible processes on to its final destination, the assembly floor.

Listing 1 shows the computer-code equivalent of a traditional conveyor loop. While the loop construct itself is universal, its design commits it to forever counting fenders. To reuse this code in an averaging operation, it would have to be cloned and modified.

Listing 1. Code for Counting Fenders

```
SET(PRO:PartKey,PRO:PartKey)
LOOP
  NEXT(Product)
  IF ERRORCODE() THEN BREAK.
  IF PRO:Part <> ‘FENDER’ THEN BREAK.
  GLO:Count += 1
END
```

Now consider the more abstract loop construct in Listing 2.

Listing 2. Abstracted code to perform a variety of checks.

```
SELF.This = SELF.That
SET(SELF.Key,SELF.Key)
LOOP
   NEXT(SELF.File)
   IF ERRORCODE()THEN BREAK.
   IF NOT SELF.CheckThis() THEN BREAK.
   SELF.DoSomeThing()
END
```

While the traditional code is clear, concise and understandable, the abstract code is basically nonsense until I define the data context in which it operates.

Assuming that THIS and THAT are references to data entities and the method calls contain some fairly abstract (and perhaps, ultimately overridden) code, I can clarify the situation by providing an operating context that mimics the functionality of the more traditional code construct.

Listing 3. Providing an operating context by assigning references.

```
Obj.This &= PRODUCT.Part
Obj.That &= TYPES.PartType
Obj.That  = ‘FENDER’
Obj.File &= PRODUCT
Obj.Key  &= PRODUCT.PartKey
```

The original OBJ.CheckThis() function may quite conceivably have been prototyped as a *placeholder* virtual method that does nothing but return True or False. Placeholder methods are used frequently in OOP design. They are meant to be overridden and provide hooks whereby functionality can be readily added to the original design without modification of the original code.

Listing 4. A placeholder method.

```
Obj.CheckThis   PROCEDURE ()
  CODE
  RETURN False
```

In the context that I want to use this method I have chosen to override it and introduce some of my own code.

Listing 5. A method derived from a placeholder.

```
Obj.CheckThis   PROCEDURE ()
  CODE
  RETURN CHOOSE(SELF.This = SELF.That, True, False)
```

Assume that the `OBJ.DoSomething()` procedure was designed as a counter; a virtual method that increments an internal "count" property every time called. To have this method perform some other function such as averaging it could be overridden without affecting the processing loop code in any way.

Listing 6. A derived DoSomething method which counts items.

```
Obj.DoSomeThing   PROCEDURE ()
  CODE
  SELF.Count += 1
```

Both the traditional and abstract code examples now do essentially the same thing – count fenders. So besides the fact that I've made the abstract code a lot harder to read and added a bunch of initialization requirements, what have I gained?

The abstract loop code, without modification, can be used to operate on almost any kind of data. By holding off on identifying the data until run-time, I've increased the flexibility of the code by the same order of magnitude that the more abstract power and free conveyor improves the flexibility and power of a traditional overhead-conveyor.

Early and Late Binding

In the programming world this temporal aspect of when it is actually decided what data are to be pushed into which procedure is referred to as the *binding point*. You may be familiar with the terms *early binding* and *late binding*.

Early binding implies that the specific data to be processed by a procedure or method are identified early on, usually at compile time. It can also imply that the procedures assigned to process or test particular units of data are determined early on, or at compile time.

Late binding implies that the decision as to what data are to be slotted into which procedures is made some time during the course of application execution. It can also imply that there is no predetermined set of procedural processes into which these data must inexorably be directed.

Some of the screams of agony that came even from long-term Clarion users on the introduction of the ABC templates and classes were, I'm certain, precipitated by the data and operational flexibility exhibited by most OOP methods. I remember writing something to this effect in a support document for one of TopSpeed's educational classes: *"The vexing thing about trying to read and logically follow OOP code is that you must always first decide the context in which it is operating..."*

It would be equally vexing to ask a plant manager where a particular door or fender hanging from a power and free conveyor might be going next. The answer would inevitably be prefixed by "that depends". Two doors hanging side by side on the conveyor might be in fact, be going to entirely different destinations depending on their state of completion or whether they were destined for the sedan or the coupe model. In the case of the conveyor, the properties of a part are inspected each time the part passes some critical conveyor junction. A bar code or microchip identifies the part as it passes a sensor. Then a computer looks up the part's production properties and shunts it to the appropriate process or into storage.

Traditional procedural code, generally speaking, tends to be hard-wired or early binding. Just like the traditional conveyor loop, it goes from here to there in a predetermined time, reliably and predictably, but not with much flexibility. The more abstract nature of OOP code tends toward less hard wiring and utilizes more late binding of data and procedures. There's a whole lot of "that depends" incorporated into a good OOP design because the nature and condition of the data (read: properties) determine its path through the various processes (read: methods).

Inspecting Data at Run Time

Anyone who has been writing Clarion - or any computer code for that matter - for any amount of time has run into this situation. You've passed a generalized structure into a procedure and want to extract or insert specific information.

Listing 7. A procedure which receives a generalized structure.

```
ProcessThis    PROCEDURE (*GROUP xGroup)
   CODE
```

This is a dead-end. I can pass in any group, but since I don't know what fields there are in the group I can't do much with it. Here's another angle.

Listing 8. A procedure which receives a specific structure.

```
ProcessThis    PROCEDURE (*SPECIFICGROUPTYPE xSpecificGrp)
   CODE
   xSpecificGrp.Part = SELF.This
```

Now, since I've defined the group to be a specific *type* of group, the compiler can resolve the field names for me when I refer to them. This is a good example of early binding. It commits the procedure to forever having to process a specific kind of group. And now my procedure is hard wired. Like a traditional overhead-conveyor it can do a good job, but the possibility of processing other data structures is not inherent in the design.

I'm not saying this technique is something that you should *not* use. In fact, you will likely use it as often as not, for those situations where the data are fixed and predictable.

But what about those parts of my programs that have to be data flexible? What if I want to write generalized procedures that can process any data regardless of its look and feel? I want to hitch a variety of structures onto my "data conveyor" and have it come out processed correctly at the other end.

The ABC BrowseClass is an excellent example of abstract OOP. It can manipulate a *Parts* file or a *People* file with equal ease. While the C2.X templates crank out reams of hard wired code, a separate batch of repetitive code for every browse, the BrowseClass does it all from one code base. Closer scrutiny of C2.X browse code makes it clear that most of the code is the same from browse to browse, with only the variable names changing. ABC BrowseClass code is reused unaltered, while a variety of data are referenced into it and are operated on. Where deemed necessary, certain of the more abstract processes (methods) are left incomplete and are swapped out through the auspices of *overriding* and late binding.

To add basic abstractness to my procedures I can reference simple data into them using the ANY data type, saving me from having to know what's coming and hence, committing my design to use, say, a LONG or STRING before it's even compiled. But what of GROUPS and QUEUES and RECORD STRUCTURES? How can I pass them between processes without committing to any *specific* data layout? You may have come across the term "property inspection". It's what my friend's oh-so-smart power and free conveyor did to determine "who" it was carrying, "what" still needed to be done to the part being carried and "where" the part was going.

Not surprisingly, Clarion provides a number of functions whose sole purpose is to allow *inspection* of data structures at run time so that you can avoid committing your code, at compile time, to any specific structure. These are the WHO, WHAT, and WHERE functions. Like my friend's overhead-conveyor, these facilities help to make your code more *abstract*. Conversely, it also makes your code more difficult to read and more dependent on the data context - but that's a small price for the advantages gained.

Three Data Structure Interrogation Functions

When a procedure references an unknown group or record structure, as it did in the preceding example Listing 7, it needs to be able to identify the structure's components by name (WHO), by reference (WHAT), and by ordinal position (WHERE) so that these individual elements can be manipulated. Here is a record structure for illustration.

Listing 9. A file structure.

```
People      FILE,DRIVER('ODBC'),RECLAIM,PRE(PEO),BINDABLE,CREATE,THREAD
KeyID          KEY(PEO:ID),NOCASE,OPT,PRIMARY
KeyName        KEY(PEO:Name),DUP,NOCASE,OPT
Record         RECORD,PRE()
ID               LONG
Name             STRING(30)
Address          STRING(30)
City             STRING(20)
State            STRING(2)
Zip              STRING(10)
BirthDate        DATE
               END
               END
```

Using WHO()

Given the record structure PEO:Record, and a position number, WHO() returns a string containing the field name. With WHO(), you can ask, "Name the second field in the record structure." For this record structure the answer would be: 'PEO:Name'. In the example below, ThisField is passed the string value 'PEO:Address'.

```
ThisField = WHO(PEO:Record, 3)
```

Listing 10 shows an abstract procedure that illustrates WHO() in action. This method returns an SQL SELECT statement, given file and record references and will work with any file because it is not hard-wired to any specific file or record.

Listing 10. Using WHO to examine a group structure.

```
MyClass.MakeSelect PROCEDURE (*FILE xFile, *GROUP xRec)
Ndx  USHORT,AUTO
Fn   CSTRING(50)
Sel  CSTRING(1000)

  CODE
  LOOP Ndx = 1 TO xFile{PROP:Fields}
     Fn  = WHO(xRec, Ndx)
     Fn[INSTRING(':',Fn,1,1)] = '.'
     Sel = Sel & Fn & ','
  END
  Sel[LEN(Sel)] = ''
  RETURN 'SELECT ' & UPPER(Sel)
```

For the record structure illustrated in Listing 9, the MakeSelect function returns a string, as follows:

```
'SELECT PEO.ID,PEO.NAME,PEO.ADDRESS,PEO.CITY,
    PEO.STATE,PEO.ZIP,PEO.BIRTHDATE'
```

Just so that you are aware, this select statement assumes the file's PROP:Alias has been set to 'PEO'. For simplicity's sake, the code also assumes all the field names contain a colon, which may not be the case in the event there are external names in the dictionary.

When inspecting a record structure, the value returned by WHO(Record, x), is equivalent to File{PROP:Label, x}. Example Listing 11 below has identical functionality to Listing 10.

Listing 11. Using PROP:Label instead of WHO.

```
MyClass.MakeSelect PROCEDURE (*FILE xFile)
Ndx  USHORT,AUTO
Fn   CSTRING(50)
Sel  CSTRING(1000)

  CODE
  LOOP Ndx = 1 TO xFile{PROP:Fields}
     Fn  = xFile{PROP:Label,Ndx}
     Fn[INSTRING(':',Fn,1,1)] = '.'
     Sel = Sel & Fn & ','
  END
  Sel[LEN(Sel)] = ''
  RETURN 'SELECT ' & UPPER(Sel)
```

The data file described in Listing 9 contains no compound structures. On the other hand, the record structure in Listing 12 presents a special situation because it contains Clarion's representation of an SQL AnyWhere *TimeStamp* field. In the database, a TimeStamp consists of 8 adjacent bytes. The first four contain a date value, the last four a time value.

Listing 12. A record structure with an SQL TimeStampe field.

```
People      FILE,DRIVER('ODBC'),RECLAIM,PRE(PEO),BINDABLE,CREATE,THREAD
KeyID          KEY(PEO:ID),NOCASE,OPT,PRIMARY
KeyName        KEY(PEO:Name),DUP,NOCASE,OPT
Record         RECORD,PRE()
ID               LONG
Name             STRING(30)
Address          STRING(30)
City             STRING(20)
State            STRING(2)
Zip              STRING(10)
DateTime         STRING(8)                 !Only this field is in the DB.
DateTime_GROUP     GROUP,OVER(DateTime) !Dictionary import places a
DateTime_DATE        DATE                  !group over the DB TimeStamp
DateTime_TIME        TIME                  !field.
                   END
               END
             END
```

Clarion's dictionary imports a `TimeStamp` as an 8 byte string, over which it places a group structure consisting of two longs. Since only `DateTime` actually exists in the file, placing any of the `DateTime_GROUP` fields in the select statement would result in an SQL error. Groups such as this return `PROP:Over = True` when tested, so it's easy enough to have my `MakeSelect` procedure leave them out of its generated select statement. But there's an added consideration. When I build a browse I don't necessarily select all the fields in the database. If I choose to place `DateTime_Date` in my browse, the code that generates the select statement must be smart enough to select `DateTime` from the database. The function `OBJ.GetActualField()` below, will return the correct select field name when the third parameter contains the name of a field inside one of these `OVER()` groups.

Listing 13. A procedure to get OVERed field.

```
MyClass.GetActualField PROCEDURE (*FILE xFile, *GROUP xRec, STRING xFName)
Ndx   USHORT,AUTO
Idx   USHORT,AUTO
FName CSTRING(50)

  CODE
  FName = CLIP(UPPER(xFName))
  LOOP Ndx = 1 TO xFile{PROP:Fields}
  IF xFile{PROP:Type,Ndx} = 'GROUP' AND xFile{PROP:Over,Ndx} THEN
     LOOP Idx = 1 TO xFile{PROP:Fields,Ndx}
       IF UPPER(xFile{PROP:Label,Idx+Ndx}) = FName THEN
          FName = WHO(xRec,xFile{PROP:Over,Ndx})
       END
     END
    END
  END
  RETURN FName
```

Using WHAT( )

Given a group or record structure and a position number, WHAT() lets you to determine the contents of specific field in that group or record structure without knowing any field names. With WHAT() you can ask, "What value is stored in the second field of the record structure?" In the example below, the value stored in PEO:Name is copied into the string variable MyString using the simple assignment operator, an equal sign. Notice that nowhere in the code is the field label PEO:Name ever used.

Listing 14. Using WHAT to get a field name.

```
MyString STRING(50)

  CODE
  MyString = WHAT(PEO:RECORD, 2)
```

But WHAT() does more than that. It can supply a reference to the data structure field. Used with the reference assignment operator &=, WHAT() provides a way for you to assign values to and from any group field without your having to know the field's name. In the example below, an ANY variable called MyAny is referenced to PEO:Name using WHAT(). Once the reference is made, MyAny also contains the contents of PEO:Name, but not because the data were copied as in example Listing 14, but because MyAny becomes PEO:Name. Notice again, that nowhere in the code is the field label PEO:Name ever used.

Listing 15. Using WHAT to get a field reference.

```
MyAny ANY

  CODE
  MyAny &= WHAT(PEO:RECORD, 2)
```

Following the reference assignment above, a simple assignment made to MyAny is functionally equivalent to a simple assignment made to PEO:Name. Hence, the assignments below provide an identical outcome.

Listing 16. Using reference assignments.

```
MyAny = 'Bob Smith'
PEO:Name = 'Bob Smith'
```

Clarion's WHAT() function can be used to design an abstract method that's able to pass field values from any file record into any queue record, without ever mentioning the file or queue field names, or knowing exactly which file or queue we are working with. For simplicity's sake Listing 17 assumes that the queue structure contains a field of the same data type, in the same ordinal position as the record structure. Remember that this isn't really necessary, but doing so would unnecessarily complicate the example code. For the purposes of this example, we'll assume the file structure described in example Listing 9 is being passed into the BufferToQueue() method.

Listing 17. Passing field values into a queue.

```
MyClass.BufferToQueue   PROCEDURE  (*GROUP xRec, *QUEUE xQ)
Ndx     USHORT(0)
Qf      ANY
  CODE
  CLEAR(xQ)
  LOOP
    Ndx += 1
    IF WHO(xRec, Ndx) = '' THEN BREAK.
    Qf &= WHAT(xQ, Ndx)
    Qf  = WHAT(xRec, Ndx)
  END
  ADD(xQ)
```

Here a loop inspects the fields in xRec one at a time. Once WHO() returns an empty string, all the fields have been processed and the loop is broken. On the next line, the ANY variable Qf is referenced to the current queue field. Now, any data simple-assigned to Qf results in the data actually being copied into the current queue field. The simple assignment made from WHAT() to Qf on the last line inside the loop does that. After the loop runs its course, a copy of the data stored in group structure xRec has been given to queue structure xQ. This loop will work with any record/queue combination and replaces hard-wired code like that shown below, in Listing 18.

Listing 18. Hard-wired code to assign fields to a queue.

```
FillQueue   PROCEDURE  (*PEO:RECORD xRec, *PeopleQ xQ)

  CODE
  CLEAR(xQ)
  xQ.Id = xRec.Id
  xQ.Name = xRec.Name
  xQ.Address = xRec.Address
  xQ.State = xRec.State
  xQ.Zip = xRec.Zip
  xQ.BirthDate = xRec.BirthDate
  ADD(xQ)
```

The end result of these two example procedures is identical. But only Listing 17 is abstract enough to be reusable with other record/queue combinations. The Clarion ABC classes use this kind of code while Clarion 2.x templates generated hard-wired code like that in Listing 18.

Using WHERE()

Given a group or record structure and a field label, Clarion's WHERE() function lets you to determine the ordinal position of a field in the group or record structure. With WHERE() you can ask, "Where in the record structure is PEO:Name?" Remember that a reference to any group structure field is just the same as having the field label. This makes it functionally the reverse of WHAT(). Hence, WHERE(PEO:Record,PEO:Name) will yield a return value of 2. In the example below, a combination of WHO() and WHERE() returns the field name in string form given a field reference, and a record structure.

Listing 19. Using WHO and WHERE to get field names.

```
MyClass.GetFieldName   PROCEDURE  (*GROUP xRec, *? xField)

  CODE
  RETURN WHO(xRec, WHERE(xRec, xField))
```

The function in example Listing 20 can be used to test whether a given field belongs to a given record structure, group or queue.

Listing 20. Locating fields within a structure.

```
MyClass.IsFieldInRecord   PROCEDURE  (*GROUP xRec, *? xField)

 CODE
 RETURN CHOOSE(WHERE(xRec, xField) > 0, True, False)
```

Interestingly, `WHERE()` can be made to deliver two different ordinals for the position of a field when the field is inside a compound group. Using the record structure in Listing 12, the ordinal position of field `DateTime_Time` relative to `PEO:Record` is 9. But relative to `DateTime_Group` its ordinal position is 2. However, `WHERE()` returns no ordinal value for fields inside an `OVER()` structure, relative to the inner group.

In Summary

An increase in abstractness, whether in machines or in computer code, tends to create a need for more and better property inspection mechanisms.

I explained that my friend's power and free overhead-conveyor, unlike its simple-loop ancestor, is filled with sensing devices, in order to identify who is being carried, what still needs to be done to that part and where the part is going.

In computer code, abstractness results from, among other things, delaying the naming of data until run-time. This in turn leads to procedures and functions that are less hard-wired and more flexible from the standpoint of the variety of data that can be handled. The computer equivalents of sensing devices are *data structure inspection functions* like `WHO()`, `WHAT()` and `WHERE()`. These allow data structures to be parsed for meaning at run-time, saving you from having to resort to binding a specific data structure directly to a procedure, and thus reducing its reusability.

There are many more such mechanisms in the new, "OOP-ified" Clarion. Although most of these are not functions in the same sense as this useful triumvirate, they are still very useful. Have a look at some of the new file-related properties such as: `PROP:Label`, `PROP:Type`, `PROP:Components`, and `PROP:Field`. There's a gold mine there for a creative mind.

There may be fortunes to be made writing Clarion too!

Gus M. Creces is a bona fide *Clarion junkie. Prior to joining TopSpeed, he worked as an independent software developer, using Clarion, primarily in the engineering, retail and manufacturing fields. With Topspeed he taught and lectured about his favorite topic, Clarion, for two years. In the past year, Gus has worked for the Consulting Division as a Consultant/Developer.*

Vol 1, No 2
March 1999

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ

Feature Article

Debugging Without The Debugger

by Dave Harms

A Clarion Open Source Project Article

I have a guilty secret: I almost never use the Clarion debugger. It's certainly not that my code is so amazingly clean that I never have to debug my programs. It's just that I've always found the debugger awkward to use, in part because the debugger doesn't work together with the source editor. For instance, I think I should be able to set a breakpoint in a source embed and run to that point. I should be able to add watch variables from within the Application Generator. These and other features would make the debugger much more useful to me.

On the other hand, even if Clarion had a terrific debugger I probably wouldn't use it much. A tool that lets me single step through my code has limited usefulness, mainly because of my lousy short-term memory. I simply can't remember more than a few steps back in my code, and I quickly become confused as to what exactly happened to get me to wherever I currently am.

What I really find useful is a log that shows me what procedures or methods were called, and what the contents of particular variables were at certain points in the code. Ideally this log will also indent the procedure/method calls, can be displayed on-screen, or can be saved to disk.

I've written a pair of classes called `cciDebugClass` and `cciProfilerClass` to handle these kinds of basic debugging/tracing tasks, and they are available as part of the Clarion Open Source Project.

In this article I'll explain some of the underlying concepts and design ideas behind these classes. You may wish to download the class (the zip includes `cciProfilerClass` and a global extension template) so you can see the full source as you read. Click here for installation and usage instructions.

You may also want to refer to the article *The ABCs of OOP - Part 1* which uses as its example a stripped-down version of `cciDebugClass`.

> NOTE: This article discusses version 1 of `cciDebugClass` and `cciProfilerClass`. The most significant changes from version .9 to version 1 are the use of manual constructors instead of automatic constructors to avoid GPFs in handcoded projects, and the addition of a global extension template to handle the creation, initialization, and cleanup of the debugging/profiling object. For version 0.9 documentation click here.

The Debugging Class

The core functionality of `cciDebugClass` is the ability to trace messages in an internal queue and display these messages on screen upon request, or write them to a file. The queue which stores these messages is TraceQ.

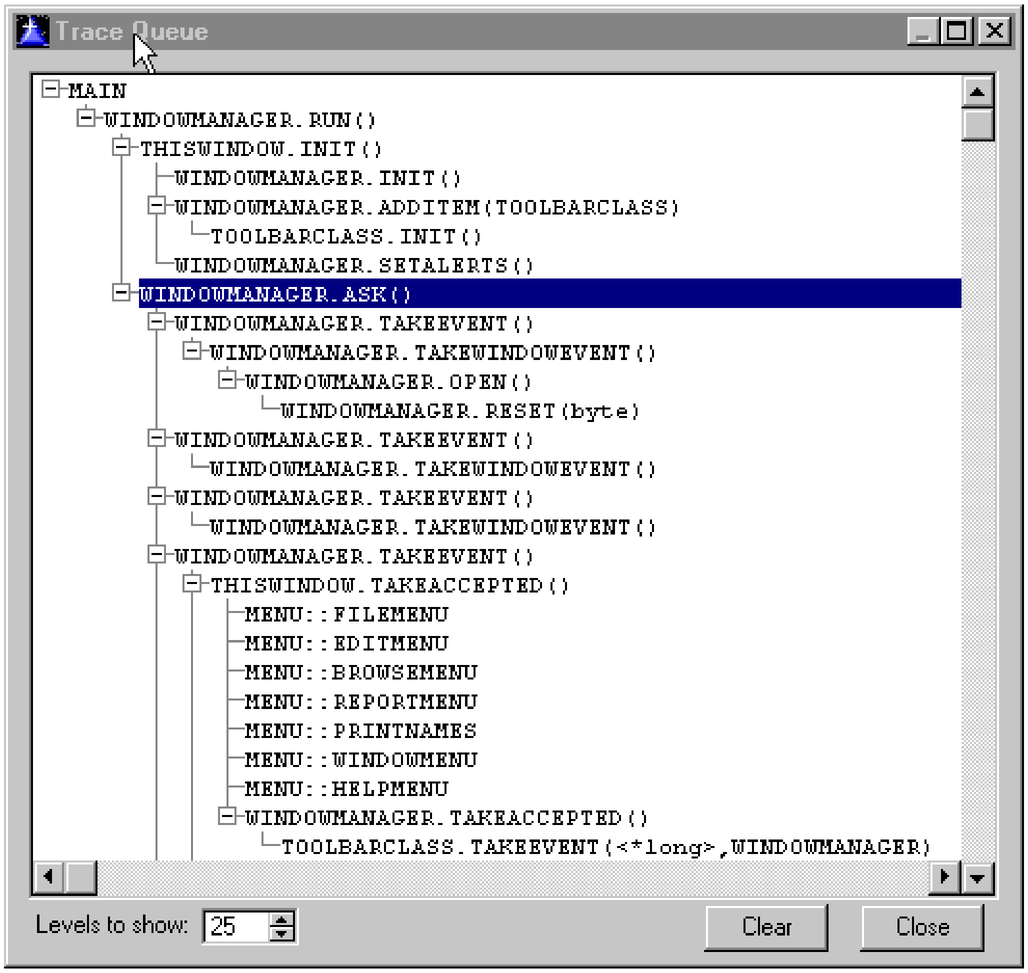
```
TraceQ &cciTraceQueue,PROTECTED
```

TraceQ is of type cciTraceQueue, which is declared as follows:

```
cciTraceQueue QUEUE,TYPE IndentLevel SHORT Text STRING(1000) END
```

The IndentLevel variable in the queue is there to help with formatting text for display, particularly when showing procedure call trees as seen in Figure 1. Although the text field in the queue is much larger than will usually be necessary this doesn't result in excessive memory usage since queue field entries are run-length compressed.

Figure 1. Displaying a procedure call tree.



```
MAIN
  WINDOWMANAGER.RUN()
    THISWINDOW.INIT()
      WINDOWMANAGER.INIT()
      WINDOWMANAGER.ADDITEM(TOOLBARCLASS)
        TOOLBARCLASS.INIT()
      WINDOWMANAGER.SETALERTS()
    WINDOWMANAGER.ASK()
      WINDOWMANAGER.TAKEEVENT()
        WINDOWMANAGER.TAKEWINDOWEVENT()
          WINDOWMANAGER.OPEN()
            WINDOWMANAGER.RESET(byte)
      WINDOWMANAGER.TAKEEVENT()
        WINDOWMANAGER.TAKEWINDOWEVENT()
      WINDOWMANAGER.TAKEEVENT()
        WINDOWMANAGER.TAKEWINDOWEVENT()
      WINDOWMANAGER.TAKEEVENT()
        THISWINDOW.TAKEACCEPTED()
          MENU::FILEMENU
          MENU::EDITMENU
          MENU::BROWSEMENU
          MENU::REPORTMENU
          MENU::PRINTNAMES
          MENU::WINDOWMENU
          MENU::HELPMENU
          WINDOWMANAGER.TAKEACCEPTED()
            TOOLBARCLASS.TAKEEVENT(<*long>,WINDOWMANAGER)
```

Initializing The Debug Class

TraceQ is a reference rather than an actual queue because queues cannot be declared statically inside a class. Instead, cciDebugClass uses a reference and creates the queue on the fly with the NEW operator. Listing 1 shows the Init() method source.

Listing 1. The cciDebugClass initialization code

```
cciDebugClass.Init                          procedure
   code
   SELF.TraceQ &= NEW(cciTraceQueue)
   SELF.ThreadQ &= NEW(cciThreadIndentQueue)
   SELF.ThreadQ.Thread = 1
   SELF.ThreadQ.IndentLevel = 1
   ADD(SELF.ThreadQ)
   SELF.NextLineToWrite = 1
   SELF.CacheSize = 50
   SELF.Disable = False
   CREATE(TraceLog)
   SELF.CollapseQ &= NEW(cciSimpleQueue)
   SELF.EventNameQ &= NEW(cciEventNameQueue)
   SELF.AddEvent('EVENT:Accepted      ',01H)
   ! Multiple calls to AddEvent omitted for brevity
```

As Listing 1 shows, `TraceQ` isn't the only queue in this class. `ThreadQ` is used to track the last indent level reached in any given thread when the program switches to a different thread. This way the class can resume logging messages at the appropriate level when the thread is resumed. The `Init()` method creates one record in the `ThreadQ` for the main program thread, and starts the indent level at 1.

Next the `Init()` method sets some caching properties. Although the class does create a log file, it doesn't write every trace message out immediately on receipt. That would waste a lot of I/O and greatly slow any program being debugged. Instead it caches as many messages as specified by the `CacheSize` property. The `NextLineToWrite` variable keeps track of the first unwritten line.

The call to `CREATE(TraceLog)` effectively empties any existing log by creating a new file called *trace.log*. The file itself is declared outside the class definition but inside the class source file. Files cannot be declared inside a class structure, although they can be declared inside a method. Another option here would be to have a reference to a file (and the single field of the file) and initialize it accordingly, but as there is only one method that writes records to the file it's probably easier just to derive a class and override that method if you want to change the way the log is written.

After the trace log is created several additional queues are created. `CollapseQ` has to do with the class's ability to display a trace in tree view, as shown in Figure 1. That trace was created by `cciProfilerClass` which is derived from `cciDebugClass`, and shows all of the ABC method calls. For many procedures, particularly browses, there are some methods which are called numerous times and which seldom have any embedded code. All messages which are added to `TraceQ` are first checked against the strings stored in `CollapseQ`, and if there's a match the indent level is given a negative sign which causes the tree to contract at that level.

`CollapseQ` was added to reduce the clutter of logs created by the derived `cciProfilerClass`. Ordinarily something added solely for a derived class would go in the derived class, not the parent class, but it seemed plausible that someone might want to have this capability even when not using `cciProfilerClass`.

The final queue created by the `Init()` method is `EventNameQ`. This queue is used to store a list of event numbers and their corresponding names. You can translate an event number using the `GetEventName()` method. If you wanted to see what events are being generated in a procedure, put the following code at the top of the accept loop (assuming you have a debug object called dbg).

```
dbg.Trace(‘Event ‘ & GetEventName(EVENT())
```

The global extension template that accompanies the debugging/profiling classes can add this code to all of your procedures automatically, if you wish, thereby giving you a complete log of all program events.

You might wonder why `GetEventName()` does a lookup in a queue instead of using a large case or execute statement. Although the standard Clarion event numbers are fixed, you can create and post your own events (using numbers from `EVENT:User` on up). The `AddEvent()` method provides a standard means of informing the class of the events it needs to track, and it also allows you to override the standard naming of the Clarion event codes without having to recompile the class.

Other Properties

Some class properties are given default values in the declaration rather than in the `Init()` method. `AutoCollapse` is a flag that indicates whether a check should be made against entries in `CollapseQ`. A false value will improve performance slightly. `CurrThread` simply tracks the current thread number. This value is checked against the value returned by the `THREAD()` function on each call to `Trace()` if the `ThreadAware` property is true. `IndentLevel` keeps track of the current indenting, and `IndentSpacing` specifies how many spaces to use per indent level. `IndentSpacing` comes into play if you choose a non-tree display of the trace log, and when the log is written to disk.

`LevelsToShow` specifies how many levels of indent should be shown expanded. This does not override level collapsing via `CollapseQ`. `StayOnLastLine` applies only to the toolbox view (see `ShowTraceToolbox()`, below) and if true forces the list box cursor to the bottom of the list whenever a new message is logged. Warning: this can have the unwanted side effect of always forcing the focus to the toolbox’s list control!

`TreeView` is a flag that indicates if the trace toolbox displays records in tree view or in an non-tree view with space-indented lines.

Methods

I’ve alluded to some of the methods, but here’s a complete listing.

```
AddEvent PROCEDURE(STRING EventName,long EventNo)
```

Ordinarily only used during initialization to load the standard Clarion event numbers and their corresponding names.

```
AddCollapseString PROCEDURE(STRING Collapse)
```

Not used in initialization of this class, but there for the use of `cciProfilerClass` which is a derived class. Specifies strings which are to be automatically collapsed when the trace is displayed in tree view.

```
FindEvent             PROCEDURE(LONG EventNo),BYTE,PRIVATE
```

Used internally to retrieve the EventNameQ record for a given event number.

```
FormatQueue           PROCEDURE(BYTE TreeView=False)
```

Switches the display queue between tree view and space-indented view. The trace log is always written in space-indented form, which includes a three digit indent number on the left side of each line. Note the use of the omittable `TreeView` parameter with a default value of `False`.

```
FormatText            PROCEDURE(STRING Text,BYTE TreeView=False),string
```

Formats a single line of text for either tree view or space-indented view.

```
GetEventName          PROCEDURE(LONG EventNo),STRING
```

Returns the name of the specified event.

```
GetThread             PROCEDURE(LONG Thread)
```

If the `ThreadAware` property is true, this method detects thread changes and records the change in the trace. It also creates new `ThreadQ` records as needed and maintains indent levels on a thread-by-thread basis.

```
HideTraceToolbox      PROCEDURE
```

Hides the trace toolbox window (see `ShowTraceToolbox()`).

```
Init                  PROCEDURE
```

Initializes the class. In earlier versions this was an automatic construct method.

```
Kill                  PROCEDURE
```

Cleans up data created by the class. In earlier versions this was an automatic destruct method.

```
SetIndentLevel        PROCEDURE(SHORT IndentLevel)
```

Specifies the current indent level. This is provided only so that user can override indent levels if desired.

```
ShowTraceToolbox      PROCEDURE
```

One of two ways to display trace logs on-screen, the other being `ViewTrace()`. `ShowTraceToolbox()` starts a display procedure on a separate thread. This isn't quite as easy as you might think. You can't simply `START()` a method because the first parameter of every method is in fact the class itself. (A side effect of this is that with methods you must always account for the hidden class parameter when using `OMITTED()` to check for missing parameters.) The way around this is to define a normal non-class procedure and call it. See `TraceToolbox()` for more.

```
Trace                 PROCEDURE(STRING Text)
```

This is the workhorse method which adds a message to the trace queue. It also calls the WriteTrace() method whenever the cache is full.

```
TraceIn                 PROCEDURE(STRING Text)
```

Increments the indent level and calls Trace(). Should be called when entering a procedure. cciProfilerClass can do this automatically for you.

```
TraceOut                PROCEDURE(STRING Text)
```

Decrements the indent level and calls Trace(). Should be called when leaving a procedure. cciProfilerClass can do this automatically for you.

```
ViewTrace               PROCEDURE
```

Displays an application modal window with the contents of the trace queue.

```
WriteTrace              PROCEDURE,VIRTUAL
```

Writes cached messages to the trace log. As this is a virtual method you need only to override it to change how the log file is written.

```
TraceToolbox            PROCEDURE(STRING Addr)
```

This procedure isn't part of the class, but is prototyped in the local map at the top of the class source file. The whole idea of a trace toolbox is to be able to see messages on screen as they are added, and since you can't start a class method you have to use a regular procedure.

Of course it isn't quite that simple. The procedure still needs to have some knowledge of the class, or it won't be able to display the queue which is inside the class. Furthermore, the procedure is an actual code entity that will be run. The class isn't. It has a TYPE attribute which means that no memory is allocated to it. You have to create an instance of the class to use it, but you have no way of knowing what that instance will be. The instance of cciDebugClass is going to have to provide that information to TraceToolbox() at runtime.

The solution is to create a reference of type cciDebugClass inside the TraceToolbox() procedure:

```
dbg &cciDebugClass
```

The ShowTraceToolbox() method passes its own address (which is determined at runtime) as a parameter to START():

```
SELF.ToolboxThread = START(TraceToolbox,25000,ADDRESS(SELF))
```

START() only accepts string parameters (up to three), so the address, which is a long, is going to be cast to a string. TraceToolbox() contains a LONG which receives the address. Clarion's automatic type conversion handles the cast.

```
DebugObjAddress = Addr
```

One last step assigns the LONG address to the reference:

```
dbg &= (DebugObjAddress)
```

Note the use of parentheses to evaluate the LONG variable. Without these you'll get an illegal reference assignment, since &= expects either a function return value or an appropriate data type.

From this point on TraceToolbox() simply references its own dbg reference whenever it needs to work with something from the debug object. The code

```
?list{PROP:From} = dbg.TraceQ
```

sets the list box's FROM property to the object's trace queue.

The Profiling Class

When I wrote my first non-OOP trace library I wanted the log to show when I entered and left procedures. I did that by embedding code at the start and end of each procedure, and it was a tedious business at best. Happily Topspeed provided a means of automatically generating procedure calls in Clarion version 4, though that probably wasn't their intention.

Since version 4 Clarion has shipped with a file called profile.clw, which you can find in your libsrc directory. This file contains a very bare-bones example of how you can intercept the start and end of each procedure (and routine). From this source you can see that the intent is to allow you to profile how much time is used by each of your methods. This capability is in the works for cciProfilerClass, but at the moment it simply provides a way of logging method calls which cciProfilerClass exploits.

To generate automatic method call logs, choose Project|Edit from the main menu, click on the Properties button for the entire application, and enable full debugging. On the Defines tab enter

```
profile=>on
```

to turn on profiling for the entire application. You'll also need to turn off profiling for the debugging and profiling classes – this is explained in more detail in the usage instructions.

When debugging and profiling are both turned on, your application at runtime will look for the following two procedures:

```
EnterProc(UNSIGNED Line,*CSTRING Proc,*CSTRING File),NAME('Profile:EnterProc')
LeaveProc(),NAME('Profile:LeaveProc')
```

If these procedures have been linked in, they'll be called each time your application enters or leaves a procedure or routine. cciProfilerClass uses these profiling hooks to log corresponding messages to the trace queue via the ProcStart and ProcEnd methods.

ProcStart and ProcEnd

ProcStart does a bit of massaging of method names to change them from the format

```
method(class,param1,param2)
```

to the more familiar

```
class.method(param1,param2)
```

`ProcStart()` also does a bit of fiddling with routine calls. It's unfortunately the case that when you return from a routine, you get a `ProcEnd()` on the procedure but not on the routine. Since `cciDebugClass` keeps a stack of indent levels, this can result in mismatches in the log. It gets worse if there have been a succession of routine calls (each with a `ProcStart()`) before the return. `ProcStart()` gets around this by using a hard coded list of the problem ABC routines.

`ProcEnd()` simply decrements the indent level, and that's about all there is to say about the methods except that `Init()` also makes a few calls to `CollapseString()` to avoid cluttering up the trace with repetitive calls (although the disk log will contain *all* the trace messages).

Exploring ABC with cciProfilerClass

You can use `cciProfilerClass` to get a window on how your ABC (or legacy) application functions. Following the usage instructions set up your application for profiling, and call the `ShowTraceToolbox()` method from a suitable menu option or embed point. With the trace toolbox active you can see the method calls onscreen as they happen, along with any trace statements you've inserted. If you're using the global extension template, go to the Options tab and check Record All Events. This will add messages recording all windows events to the log so you can see how ABC responds to user actions.

Tracking GPFs

You can use the profiling class to quickly track down GPFs. Set up your application to do procedure as described above, and on startup set the `CacheSize` property to 1. That will force each message to be written out to `trace.log` immediately. Let the program run to GPF, and then examine `trace.log` to find out the last procedure/method call which succeeded. From this point you can use the debugger or additional trace statements to uncover the problem.

Possibilities

There are a number of enhancements I'd like to see added to these classes. As I've mentioned, the original intent of the `EnterProc()` and `LeaveProc()` procedures is to enable procedure profiling which can provide detailed information about how much time is spent on individual methods on a per-call and total calls basis, and on how many times a given method is called. This is invaluable when you have performance issues to address. cciProf.clw already contains prototypes for 16 and 32 bit API functions used to get system clock times with a much finer increment than the 1/16 of a second afforded by the `CLOCK()` function.

`cciDebugClass` could also be enhanced with watch variables by means of a queue of ANY variables with the value displayed on the trace toolbox, or reported along with each trace message, or on a specific method call, or only when it changed.

You're free to modify these classes and distribute the modified versions, but keep in mind that these classes are (as of version 1) covered by the Developers' Open Source License, and all changes you make are automatically covered by that license. If you've made changes you'd like to see included in the Clarion Magazine COSP distribution of these classes, please send them to cosp@clarionmag.com.

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, *published by SAMS (1995). His company, CoveComm Inc, publishes Clarion Magazine.*

Subscriber Services

Free User Group Newsgroups

If you're interested take our poll & let us know!

Clarion Magazine provides free newsgroups to Clarion User Groups. These newsgroups are for non-profit activity only. You can use them to provide a chat area, for support, or to organize group projects.

These newsgroups are private to your user group. Clarion Magazine reserves the right to monitor newsgroups and to terminate newsgroups without notice.

Only the contact person needs to be a subscriber to Clarion Magazine. The CUG may distribute the user id and password to all user group members, or to members of other CUGs.

Please allow one week for your application to be processed.

Contact Person

Name	
Email address	
Contact person's Clarion Magazine User ID	

User Group Information

User Group Name	
Location (City/State/Country)	

Newsgroup Information

You can have up to five sub-newsgroups in addition to the top level newsgroup. For instance, if you're applying from the East Overshoe CUG, and your top level newsgroup is called *overshoe*, you could set up a structure something like this:

overshoe.news
overshoe.projects
overshoe.projects.special
overshoe.support
overshoe.chat

Top level newsgroup	(This should be an abbreviation of your user group name).
Newsgroup User ID	(All cug members will use the same user id. This should not be the same as the contact person's user id and it cannot be the same as the user id for the clarion magazine newsgroups.)
Newsgroup Password	
Sub-newsgroup #1 (optional)	
Sub-newsgroup #2 (optional)	
Sub-newsgroup #3 (optional)	
Sub-newsgroup #4 (optional)	
Sub-newsgroup #5 (optional)	
Comments	

Submit Reset