e-Xtremé Caché Objects

By Paul Gausden, George James Software

The ‘Generator’ option for a Method in Caché Objects is a feature rarely used by programmers. Examining their usage in the %Library package of classes soon discourages the viewer. They make extensive use of macros (many of which disguise the fact that InterSystems (ISC) are using direct global access in their library objects) and routines with hidden source code.

However, this feature is one of the most useful for adding new behaviour to Caché Objects where none existed before, and yet, ISC do not provide any documentation on this powerful feature. Also, ISC have not promised that the macros used will not be changed in future releases, though their own code relies very heavily on them, so this is unlikely. They also use the macros to lessen the impact of their own internal changes to Caché, so your code will stay up to date – some of my generator methods have worked without change since Caché 2.1.6.

A lot of the “observations” on the intersys.public.cache newsgroup are about missing functionality in Caché Objects. A couple of good examples, which can be added via generator methods, are:

- Retaining the stored value of a persistent object (for auditing changes when saving)
- Opening an object via one of it’s unique indices instead of it’s OID

Without generator methods, you would have to write similar code in all the classes where the functionality was required. By using the methods built into special “helper” classes (a class added to the list of super classes that just adds functionality) or by extending other super-classes with these methods (e.g. Persistent), you only need to write the code once. These classes can be re-used in many other classes as well as future projects. It is worthwhile building up a library.

ISC have always said that Caché Objects is extensible – why should they write this functionality when is it bound not to be generic enough for everyone?

Apart from the uses listed above, other potential uses I have come across over the years for these methods are:

- To expose more information about a class as class methods.
- To add more flexible methods to data types.
- To write a custom (non-SQL?) query system.
- As a custom storage class – not an easy task, as the compiler is quite tightly bound to the ISC storage classes (unless of course you want to write the compiler too).

What are Generator Methods?

A generator method is code that writes code. The code you write is run at compile time to generate code for another method that is run at run time.

A small note here, the method signature defined for the generator is used as the method signature for the compiled code (although this can be overridden if you know the appropriate macros). Therefore some thought needs to go into the parameters your final method will need.

As a small example, the following method generates a list (as a $List) of all properties in a class as a class method expression:

```
n p
s %code=0
s %codemode=$$$cMETHCODEMODEEXPRESSION,%code=""
s p=""
f  s p=$o($$$PROP($$$MGINFO,%class,p)) q:p=""  DO
```

```
. s %code=%code_""""_p_""","
i %code'="" s %code="$lb("_$e(%code,1,$l(%code)-1)_")"
QUIT $$$OK
```

Where to start?

Start with the end. Write out what you want to end up with, the code you require if applied to a particular class. Then do the same for another class – (you just did what the generator method will be doing).

I'll start with an easy and extremely useful example:

Caché Objects does not provide an easy way of determining what is the set of valid values for a property when a DISPLAYLIST or VALUELIST parameter is defined for the property.

In your CSP page or VB front end, you are going to want to populate combos and lists with these values before you have an instance of the class open (starting to sound a bit like a class method is needed) and you don't want to keep changing the front end if you decide to add another possible value to the list.

You don't need to use DISPLAYLIST's, you could just use another class to store the list, but these small classes just add to the clutter in your nice clean object design.

What you need to know before you start coding…

There are a few of those ISC macros you need to know about. All are stored in the %occ*.INC include files.

$$$GENERATE(CODE) from %occCodeGen.INC is the most used. It stores the line of code for the compiler to save as the run time method code.

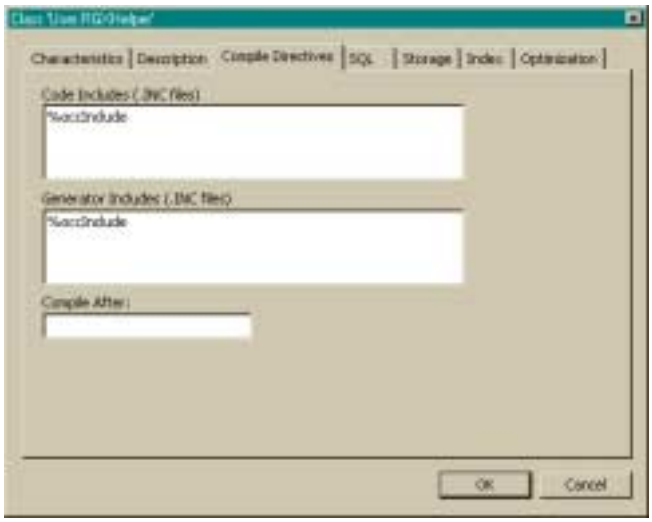
All the macros defined in %occStatus.INC are useful - $$$OK, $$$ERROR, etc and there are a few helpful macros that generate loop code for various items in a class in %occConstant.INC

However the largest set of macros, which give access to everything (and then some more!) you need to know about a class (the meta-data – stored in ^oddDEF, ^oddCOM etc), are in %occReference.INC and %occReference2.INC. With these you can do "illegal" object programming, such as accessing private properties and getting at the storage mapping information (for the supplied storage classes) – but not in this article.

Commonly used macros for getting at the class meta-data are:

$$$CLASS*	e.g. $$$CLASSpersistent	Class properties
$$$PROP*	e.g. $$$PROPcalculated	Property properties
$$$PROPPARAM*	e.g. $$$PROPPARAMdefault	Property parameters
$$$PARAM*	e.g. $$$PARAMdefault	Class parameters

To make use of all these include files, they must be added to the lists of Code Includes and Generator Includes for a class (under the Compile Directives tab of the class properties in the Object Architect). Rather than having to remember all the names and add them all, ISC have conveniently bundled them all into one INC file - %occInclude.INC



Next, you need to know what is available to a generator method when it runs at compile time.

There are a few variables passed to a generator method, these include:

```
%class,%property,%method,%parameter,%codemode,%code
```

%class – name of the current class being compiled.

%property – name of the current property, **but only if the generator is in a data type class or a referenced class – i.e. not available in class methods.**

%method – used to access information about the current method being compiled.

%parameter("<parameter name>") – array of property parameters if %property is not ""

%codemode – defaults to "code", but can be "expression" or "call" (or for the really extreme programmers out there "generator") - $$$cMETHCODEMODE* macros exist for these keywords

%code – and array of the generated code lines (usually updated from within the $$$GENERATE macro), except for generated expressions which set %code="expression"

Finally, you should always exit the method with Q $$$OK, or you'll get an error during compilation. That is, unless you need to force an error when the designer has not correctly specified the class.

Coding

We now have all the pieces to write a class method to access the DISPLAY/VALUELISTS. In this case, we want to pass in the property name and get back a list containing the allowed values.

Using this information, start to substitute the parameters into the code you started with, to make it more generic. Here we should also assume that we can get at the properties of the class or properties of properties and their parameters.

If non-generic parameters are left over in the code, then we have our first problem. This information can be supplied to the generated (run-time) method as parameters, or to the generator method (compile time) as a fixed parameter in the class to be compiled, set by the class designer. The two examples use both methods; it depends on how "static" the information is.

I usually create a separate class and group similar generator methods within them, so they can be used as a second super-class, rather than sub-classing. It allows slightly more flexibility as to where the methods are used, particularly if you have a library (Package?) of such classes.

So the steps now are:

1. The code for the class method I want to end with is something like this for one of my classes:

```
getList(Property)    ;
     I Property="ComputeAlways" Q $LB($LB("Y","Yes"),$LB("N","No"))
     I Property="DataType" Q $LB($LB("TEXT","TEXT"),$LB("DATE","DATE"),$LB("TIME","TIME"))
     I Property="Password" Q $LB($LB("Y","Yes"),$LB("N","No"))
     I Property="Protection" Q $LB($LB("A","Add"),$LB("AE","Add/Edit"),$LB("R","Retrieve"))
     I Property="Required" Q $LB($LB("Y","Yes"),$LB("N","No"),$LB("M","Maybe"))
     I Property="SearchKeyHit" Q $LB($LB("!","!"),$LB("!*","!*"))
     Q ""
```

I need a line for each property that has a display list.

2. Create the class e.g. GEN.GetList (usually as an abstract class as there are unlikely to ever be any instances of this)
3. Add '%occInclude' to the class compiler directives.
4. Create a method called getList(), define it as a class method and select the generator option box. My generic code required one parameter 'Property' to determine which list to return. The return type is set to %Library.List for this particular case.
5. Write the code … the $$$PROP and $$$PROPPARAM macros can be found in %occReference, $$$MGINFO compiles to ^oddMAC and the #; comments are stripped out by the MAC compiler.

```
; Return display/value pairs for properties with display/value lists
; Returns a list of 2 item lists for a property
; each item contains the value & corresponding display value
; if no display value, second item = first item
; if no value list, first item is ""
;
; Class Method
; requires property name as parameter
;
s %code=0
N P,codeLine,dispList,valList,DDelim,VDelim,S,SDEL,KEY
$$$FOREACHproperty($$$MGINFO,%class,P)  D
#;
#; ignore certain types of property
. If $$$PROPprivate($$$MGINFO,%class,P) Q
. If $$$PROPmultidimensional($$$MGINFO,%class,P) Q
. If $$$PROPcalculated($$$MGINFO,%class,P) Q
#;
#; DISPLAY AND VALUE LIST parameter for property P ?
. S dispList=$G($$$PROPPARAM($$$MGINFO,%class,P,"DISPLAYLIST"))
. S valList=$G($$$PROPPARAM($$$MGINFO,%class,P,"VALUELIST"))
#;
#; Do we have a list ?
. I dispList="",valList="" Q
. S VDelim=$E(valList),DDelim=$E(dispList)
#;
#; Find longest list
. S codeLine=" I Property="""_P_""" Q $LB("
. S S=valList,SDEL=VDelim I $L(dispList,DDelim)>$L(valList,VDelim) S S=dispList,SDEL=Ddelim
#;
#; For each element in the list generate:
#;   $LB(valueListItem,displayListItem)[,]
. F I=2:1:$L(S,SDEL) D
. . S KEY=$P(dispList,DDelim,I) I KEY="" S KEY=$P(valList,VDelim,I)
. . I KEY'="" S codeLine=codeLine_"$LB("""_$P(valList,VDelim,I)_""","""_KEY_""")"
. . I I<$L(S,SDEL) S codeLine=codeLine_","
. ;
. $$$GENERATE(codeLine_")")
#; by default Q "" is automatically generated at the end
Q $$$OK
```

The method ends up in the Package.Class.Gn routine created by the compile. Also bear in mind that the method is also run when the abstract class that contains it is compiled – your code may have to check it's current context e.g. is this method only valid for persistent classes.

Another Example – Retaining Stored Values

This functionality is useful for auditing changes to instances of persistent objects that are opened for any length of time.

The aim is to provide a method that detects changes in some or all properties (or even methods with return values) in the current instance vs. the stored value, at the time %Save() is run, and to add this to a generic audit class.

To achieve this we need to save the stored values somewhere.

One solution is to use a private multi-dimensional property in the class that contains the generator method (the helper class) and store the initial values as:

```
auditStored("<Property Name>")=<Stored Value>
```

This property is then inherited into your class and can be ignored for the rest of the design work.

Similarly, if we want to target specific properties in a class, we can add a parameter to the helper class, also inherited, for the class designer to override with a list of properties to be audited.

We require two generator methods. One to load the stored values into the array, when an object is opened and the other to update the audit on saving (and reset the stored values in case it is saved a second time).

These could be %OnOpen() and %OnAfterSave() (not %OnBeforeSave(), as you might not have an OID yet if it is a new object), but if hand-coded methods exist in the class, they would overwrite our methods. Alternatives might be to

- Write another (non-generator) method in the helper class that calls %Open, runs the auditStart() method and returns the open object, similarly a method to close the audit and save the object.
- In the client code, just call the open, then call the auditStart() method (similarly for the save)

After adding the private, multi-dimensional property "auditStored" and the parameter "AUDITPROPERTIES", which we shall assume will be a comma delimited list of properties added by the class designer, to the helper class. Here are the 2 methods:

auditStart() – Instance, Generator method

```
#; Generate code to store values for audit record.
Set %code=0
#; remember this code is also run for the helper class
Q:'$G($$$CLASSpersistent($$$tLIB,%class)) $$$OK
N P,I,Z
S P=$G($$$PARAMdefault($$$MGINFO,%class,"AUDITPROPERTIES"))
I P="" Q $$$OK
F I=1:1:$L(P,",") S Z=$P(P,",",I) I Z'="" D
. $$$GENERATE(" S i%auditStored("""_Z_""")=.."_Z)
q $$$OK
```

That was quite short. All it does is generate several Set commands e.g.

```
S i%auditStored("Name")=..Name
S i%auditStored("Rank")=..Rank
S i%auditStored("Number")=..Number
```

For the second method, I have assumed a generic auditing class, which is indexed by date/time, class and oid and contains an array of the changed properties, but you could modify this to fit your auditing strategy.

auditSave() – Instance, Generator method

```
#; Generate code to create audit record for the class
Set %code=0
Q:'$G($$$CLASSpersistent($$$tLIB,%class)) $$$OK
```

```
	N P,I,Z,C
	S P=$G($$$PARAMdefault($$$MGINFO,%class,"AUDITPROPERTIES"))
	I P="" Q $$$OK
	$$$GENERATE(" N qa,mod")
	#; build a new instance of the audit record
	$$$GENERATE(" s qa=##class(GJGEN.Audit).%New()")
	#; Generate a line for each property - only update if changed
	F I=1:1:$L(P,",") S Z=$P(P,",",I) I Z'="" D
	. $$$GENERATE(" I $g(i%auditStored("""_Z_""","""_Z_"""))'=.."_Z_" S mod=1 _
d qa.auditData.SetAt(.."_Z_","""_Z_""")")
	$$$GENERATE(" I '$d(i%auditStored) S qa.newRecord=1,mod=1")
	# ; Nothing modified ?
	$$$GENERATE(" I 'mod d qa.%Close() Q")
	# ; Update other pieces
	$$$GENERATE(" s qa.auditID=..%Id(),qa.auditClass=..%ClassName(),qa.auditTime=$h")
	$$$GENERATE(" d qa.%Save(),qa.%Close()")
	#; don't forget to reset your stored values here…
	$$$GENERATE(" d ..auditStart()")
	q $$$OK
```

(**Note** – I'm not bothering to check the %Save() value for the audit record here, in practice you would)

Neither of these examples is particularly large, but both add functionality to your objects that can be re-used many times.

Finally, what you cannot do…

There are a few things you can't do with generator methods. I'll just list a few of them here…

- Override the Get/Set methods of data types – Caché objects always uses the methods stored in %Library.CacheProperty, a hidden class
- Override the LogicalToODBC() methods with generator methods – Actually, you can do this, but the ODBC access does not use the generated code. It calls the original method stored in the datatype class – a bug in my opinion.
- Add new methods at compile time.

Resources

Over the past few months, I have posted several other CDL's for generator methods to the Caché newsgroup under the threads:

- Not Numeric OID - 26th Jul 01, my reply 30th July – open an object using an index.
- Property by Name - 6th June 01, my reply 8th June – get/set properties using variable names.
- TDate data type - 14th May 01, my reply 22nd May – additional data type method.

This document and a few example CDL's are available at www.georgejames.com.