OO For The Elderly

Stepping Sideways Towards Using Objects in Applications

Dick Bowman (dick@dickbowman.org.uk)

Introduction

APL, like many APLers, has been around a long time and many successful applications have been built using time-honoured approaches. The arrival of object-oriented extensions has opened new avenues, but established APLers may not adopt these new language features without some sort of struggle.

In some ways this parallels the battles that were fought with the arrival of nested/heterogeneous arrays two decades ago, where established methodology appeared to be threatened by the new.

The idea of this presentation is to explore Dyalog's OO features through a back door, following a path taken by one elderly APL bigot – it may interest others who have a similar background, or newcomers wanting to make fullest use of APLs functionality.

The presentation focusses on "what" rather than "how" and does not set out to present Dyalog's OO features systematically, rather some of the ways that they can be used in application development. The scenario is "traditional desktop application", and code for a small application is included (hopefully not the personal data, though).

The idea is also to avoid the conventional software engineering "OO is best" brainwashing approach.

⎕NEW replacing ⎕WC, and so forth...

Windows was something of a watershed for application development, and Dyalog's introduction of `⎕WC/WG/WS/Whatever` in the late 1990s opened the door for "conventional-looking" applications to be developed in APL without the developer needing to explore the dark underbelly of writing Windows code (no need to understand, or even read, Petzold's book – and some contrast to what APL*PLUS first made us do).

It was straightforward, after a while it became almost a mechanical process, and maybe a little tedious. You remember how it went...

```
  'f' ⎕wc 'form'
  'f.1' ⎕wc 'Label' ('Caption' 'Hello World')
```

But it felt a little "clunky" - there were too many character strings, and the important stuff (what the user saw, for example) was something of a side-effect for the APL code (`⎕WC` didn't "make something" in the sense of being a function with a result – well, there's a shy result to `⎕WC` but I've never seen it used).

When dot-syntax started arriving in Version 9 , we found ourselves with a new way to do "the same old stuff".

```
  f←⎕new 'Form' ''
   f.1←f.⎕new 'Label' ((,⊂'Caption' 'Hello World'))
```

Now some of the traditional values of APL return to centre-stage.

We're making new "things" and retrieving values in a direct cause-and-effect paradigm. Instead of a "different world" where side-effects rule we've advanced by gaining simplicity – these GUI "objects" are beginning to behave like APL has for decades.

Contrast

`'f' ⎕wc 'Form'`
`'f' ⎕ws 'Size' (20 20)`
`⎕dq 'f'`
	`f←⎕new'Form' ''`
	`f.Size←20 20`
	`f.Wait`

Tidier, less quotes, more obvious – it just makes more sense (and it's closer to “normal language” - who ever dequeued for a bus?). Been writing almost all of my GUI stuff this way for over five years.

Unfortunately there are a few loose ends, where it is necessary to fall back into the old ways; Tabbed subforms are one example, `OCXClass` is another; it would be good to have these brought into line. But – the gain seems worth the occasional pain.

Building Your Own GUI Controls

The trouble with writing code for GUIs is that it's so verbose, and it's also tedious to adhere to (or enforce) consistent standards like “all forms are defined in pixels”, “all buttons are 23 by 75 pixels”, “all edit controls need a label just above them”.

On the bright side, being able to create new controls on-the-fly brings us a freedom and flexibility that seemed to be missing when I last looked at Visual Basic (both in the product and in the programmer mindset) . As with so much – it's a liberation that can be over-indulged, seen some horrifically complex dynamic form-building.

Salvation from a lot of this tedium came our way when Version 11 let us define our own classes extending the Dyalog GUI controls.

Example (a) – Simpler Simple Controls - `DogForm` and `DogButton`

`DogForm` is a variant on the inbuilt `Form` control, requiring the programmer to supply only caption and size, it assumes that coordinates are in pixels.

```
:Class DogForm   : 'Form'
⍝ Argument is Caption Size
:Field Caption←'Dogon Form'
     ∇ Create args
      :Access Public
      args←(⊂¨'Caption' 'Size'),∘⊂¨args
      :Implements Constructor :Base args,⊂ ('Coord' 'Pixel')
      ⎕DF'[DogForm: ',Caption,']'
    ∇

:EndClass
```

Note, most of the code samples here assume `⎕io ⎕ml←0 3`.

`DogButton` is a variant on the `Button` control, again requiring only caption and position from the programmer There's an embellishment here allowing optional specification of the `Attach` property (codified down to an abbreviated form, which `translateAttach` expands).

```
:Class DogButton : 'Button'
⍝ Argument is Caption Posn
⍝            Caption Posn Attach
    :Field Caption←'Dogon Button'

    ∇ create args;caption;posn;attach;⎕IO;⎕ML
    ⍝ Create the button
      ⎕IO ⎕ML←0 3
      :Access Public
      :Select ⊖⍴⍴args
      :Case 2
          args←(⊂¨2↑'Caption' 'Posn'),∘⊂¨args
      :Case 3
          (2⊃args)←translateAttach 2⊃args
          args←(⊂¨3↑'Caption' 'Posn' 'Attach'),∘⊂¨args
      :EndSelect
      :Implements Constructor :Base args,⊂('Size' (23 75))
      ⎕DF'[DogButton: ',Caption,']'
    ∇
:EndClass
```

`DogForm` and `DogButton` can be deployed in applications almost exactly like `Form` and `Button` – although there's an inconsistency in the syntax, where the inbuilt controls still seem to be clinging to the older syntax with their quotes.

Almost as a bonus, since we're producing variants on simple controls, we still have access to all of the functionality of the original

`fold←⎕NEW'Form'(('Caption' 'Hello Old World') ('Coord' 'Pixel')('Size'(200 200)))`
`fold.pressme←fold.⎕new 'Button' (('Caption' 'Press Me')('Posn' (10 100)))`
`fold.pressme.Posn←20 50`
	`fnew←⎕NEW #.UTIL.DogForm ('Hello New World'(200 200))`
	`fnew.OK←fnew.⎕new #.UTIL.DogButton ('Press Me' (10 100))`
	`fnew.OK.Posn←20 50`

Gained – Standard things happen consistently, every `DogButton` is the same size, and we can change the size everywhere just by changing the class definition (same applies for other properties). We always could make programmers behave, but it's easier now,

Things to think about – how should our custom controls be defined? What's here suits me (sometimes) – but everyone ought to decide how they decide what's automated and what isn't.

Example (b) – Compound Controls – `DogEdit`.

`DogEdit` is a little more complex, it combines an `Edit` with a descriptive `Label`, takes (some of) the tedium out of creating `Label/Edit` pairs and makes it easier to follow corporate layout rules than to break those rules...

```
:Class DogEdit : 'SubForm'
⍝ Argument is Caption Posn Value Callback
⍝             Caption Posn Value Callback Attach
    :Field Caption←'Dogon Edit'

    ∇ Focus w
      :Access Public Instance
      :If w
          ⎕NQ'edit' 'GotFocus'
      :EndIf
    ∇

    :Property Value
    :Access Public
        ∇ z←get
          z←edit.Text
        ∇
        ∇ set w
          edit.Text←w.NewValue
        ∇
    :endProperty

   :Property Active
    :Access Public
        ∇ set w
          edit.Active←w.NewValue
        ∇
    :EndProperty

    ∇ create args;size
      :Access Public
      size←50 180
      :Select ⍬⍴⍴args
      :Case 4
          args,←⊂translateAttach'nnnn'
      :Case 5
          (4⊃args)←translateAttach 4⊃args
      :EndSelect
      args←(⊂¨(⍴args)↑'Caption' 'Posn' 'Text' 'Event' 'Attach'),∘⊂¨args
      :Implements Constructor :Base (1 4⊃¨⊂args), ('Size' size)('EdgeStyle' 'Dialog')
      label←⎕NEW'Label'((↑args)('Posn'(10 0))('Size'(20 180)))
      edit←⎕NEW'Edit'((2⌷args),('Posn'(30 0))('Size'(20 180)))
      :If ×ρ3 1⊃args
          edit.onChange←'Change'(3 1⊃args)
      :EndIf
      ⎕DF'[DogEdit: ',Caption,']'
    ∇

    ∇ opt Change msg
      ⍎opt,' msg'
    ∇
:EndClass
```

Another embellishment here, a callback on the `onChange` event lets us build in validation (or whatever else might be appropriate).

Again, simple enough to deploy in applications...

```
   f←⎕new #.UTIL.DogForm ('Using DogEdit' (200 200) '')
   f.e←f.⎕new #.UTIL.DogEdit ('Edit Me' (20 20) 'abcdef' '#.Ouch')
   f.b←f.⎕NEW #.UTIL.DogButton ('OK' (80 50))
   f.Wait
```

Something that didn't “leap off the printed page” was deciding what the base class should be for these compound controls – in hindsight `SubForm` seems to have become obvious.

An arguable bonus is that standardising on `Value` to define control content is a little easier on the user-programmer than `Text` or `Items` (or ...) depending on the specific control type.

Notice also that design decisions are becoming more complex – which properties events and methods should be built into the compound controls? The more gets built in, the more complex the definition is to write and enhance. And the converse, that the solutions based on these choices become more personal.

In practice what happens is that real-life GUIs are made up of a mix-and-match combination of Dyalog's own controls and simple/compound defined controls – sometimes there's an evolution from a rough prototype built from the higher-level defined controls to a more sophisticated interface fashioned from the provided base level.

What we've achieved so far is...

- We're comfortable populating the workspace with “things” made by `⎕NEW`
- We are defining our own “things”
- We can manipulate these “things” using consistently familiar syntax

We're ready to move on and think about “things” that aren't GUI controls.

Component Files as Objects

We all “know all about” component files, they've been around forever, but sometimes we overlook historically-imposed artifice (like tie numbers). Quietly tucked away in the Dyalog distribution there's a `ComponentFile` class...

```
     )load componentfile
C:\Program Files\Dyalog\Dyalog APL 12.0 Unicode\Samples\....
     cf←⎕new #.ComponentFile 'c:\dick\temp\oocomp.dycf'
     cf.Append 2 3ρι6
1
     cf.Append 'abcdef'
2
     cf.Count
2
     cf.Name
c:\dick\temp\oocomp.dycf
     cf.Components[2]
 abcdef
     )erase cf
```

Notice that despite the O-O propaganda saying that “objects can be reused, easy-peasy, it just happens”, the programmer needs to poke around inside the code to see what we can do and how to do it – just like in the days of subroutine libraries.

Lesson 1 – Classes don't let the author-programmer off the documentation hook, if you want someone else to use your classes you have to tell them how.

We can incorporate the supplied component file class into applications – arguably it offers a cleaner coding style because we've lost the artifice of the tie number (although, in the spirit of Dyalog's ducks, they're still paddling madly away under the surface).

And, since we're picky people, we may not agree with some of the design decisions that have been made in the class definition, which we can deal with in (at least) three ways...

- Modify the supplied code (don't really recommend this)
- Build a new class based on the supplied one (the supplied `KeyedFile` is an example)
- Use `ComponentFile` as an inspiration for a class of your own (`DogComponentFile` is an example)

Something that would be useful would be for Dyalog to detail the class definitions included in the product, with some indication whether they are intended for learning or for serious use.

Objects for Data

That's set the groundwork, we're now comfortable with using `⎕NEW` to “make things”, and setting/retrieving properties, and using methods.

How about applying this to the data within applications?

DogMoney is a simple personal accounts application, covering similar ground to Quicken and Microsoft Money. Easy enough for the experienced APLer to go into Autopilot mode and write this sort of thing using exactly the same approach as they have for years Question is, what happens if we decide to use creating this application as a learning tool for O-O techniques?

A Simple Class - Payees

What we need to know about payees is simple, their name and a unique identifier (so that we can treat name as a data value in the time-honoured database tradition).

```
:Class PAYEE
 ⍝ Payee Class

    :Field Public Name
    :Field Public Id
    :Field Public Shared LastPayeeId

    ∇ create w
      :Implements Constructor
      :Access Public
      :Select ↑w
      :Case 0
          Name←1⊃w
          LastPayeeId+←1
          Id←LastPayeeId
      :Case 1
          Id Name←1↓w
      :EndSelect
    ∇

:EndClass
```

Allowing us to create a variable containing payee information, add entries to it, and access payee values in a fairly literate way...

```
   payees←⎕new #.PAYEE (0 'Ned''s Bakery')
   payees,←⎕new #.PAYEE (0 'Millie''s Millinery')
   ⍴payees
2
   ⊃payees.(Id Name)
440  Ned's Bakery
441  Millie's Millinery
```

And it's used within DogMoney in a quite traditional APLish way. Add new payees, delete unwanted ones, and so forth. In many ways `payees` is “just another variable”, albeit with some characteristics that traditional variables don't have – characteristics that make program code just that little bit more “literate”.

A Keyed Class – Foreign Exchange Rates

Foreign exchange rates are a little more complex, we really want to get straight to the nub of “how many dollars can I get for a pound?” (as usual, I'm being optimistic here).

```
:FOREX
⍝ Foreign Exchange

⍝ Sample use....
⍝     fff←⎕new FOREX ''
⍝     fff.Values[⊂'USD']←⊂'US Dollar' 1.97 20080108
⍝     fff.Values[⊂'EUR']←⊂'Euro' 1.33 20080108
⍝     fff.Rate[⊂'USD']
⍝ 1.97
⍝

    :Field Private code
    :Field Private name
    :Field Private rate
    :Field Private date
    :Field Public Shared LastRefresh←20080108

      ∇ Create w
      :Access Public
      :Implements Constructor
      code←''
      name←''
      rate←⍬
      date←⍬
    ∇

    :Property Keyed Values
    :Access Public Instance
        ∇ z←get w;index
          index←,codeι⊃w.Indexers
          z←index{α⊃¨ω}¨⊂name rate date
        ∇
        ∇ set w;key;index
          key←⊃w.Indexers
          :If key∊code
              index←codeιkey
              (index⊃name)←0⊃⊃w.NewValue
              (index⊃rate)←1⊃⊃w.NewValue
              (index⊃date)←2⊃⊃w.NewValue
          :Else
              code,←⊃w.Indexers
              name rate date←name rate date,∘⊂¨⊃w.NewValue
          :EndIf
        ∇
   :EndProperty

   :Property Keyed Rate
    :Access Public Instance
        ∇ z←get w
          z←(codeι⊃w.Indexers)⊃¨⊂rate
        ∇
   :EndProperty

   :Property AllCodes
    :Access Public Instance
        ∇ z←get
          z←code
        ∇
    :EndProperty

∇ Delete w;sel
 :Access Public
 sel←~code∊w
 code name rate date←(⊂sel)/¨code name rate date
∇

:EndClass
```

Again, once it's figured out, the code is straightforward and easily built into the application. A personal beef here, that in Version 11 Dyalog gave us an indexing function with one hand, then with the other said we couldn't use it for keyed properties.

A Hybrid – Transactions

When exploring object-oriented programming languages like Smalltalk the avid APLer is inclined toward creating a class called something like “APLArray” and a host of methods to work on it. Sometimes working with classes within APL there's an inverse tendency to lurch back toward arrays and array methods when there might be something more purist.

Consider how we need to store transaction data – there are a lot of transactions, and each one needs pretty much the same data structure as the rest. A matrix of transactions just “feels easiest”. So, what might we put into a Transactions class?

```
:Class TRANSACTION
 ⍝ Transaction Class

 ⍝ TransMatrix columns
 ⍝ 0 id
 ⍝ 1 ref
 ⍝ 2 date
 ⍝ 3 account
 ⍝ 4 payee (payee for types D W, account for type T)
 ⍝ 5 category
 ⍝ 6 amount
 ⍝ 7 type (D W T)
 ⍝ 8 reconciled

    :Field Public TransMatrix ⍝ Just to make corrections while developing code?
    :Field Public Shared LastTransactionId


    :Property  Account
    :Access Public
        ∇ z←get
          z←3⌷[1]TransMatrix
        ∇
    :EndProperty

    :Property  Amount
    :Access Public
        ∇ z←get
          z←6⌷[1]TransMatrix
        ∇
    :EndProperty

   [ ... ]

    ∇ z←AccountBalance w;trans
      :Access Public
    ⍝ Balance for a given account (Id)
      z←+/6⌷[1]AccountTransactions w
    ∇

    ∇ z←AccountTransactions w
      :Access Public
      ⍝ Matrix of transactions for a given account (Id)
      z←(w=⎕THIS.Account)⌿⎕THIS.TransMatrix
    ∇
    ∇Create w
      :Implements Constructor
      :Access Public
      TransMatrix←0 9⍴⍬
    ∇

    ∇ Append w
      :Access Public
      TransMatrix,[0]←w
      LastTransactionId←LastTransactionId⌈⌊⌈/0⌷[1]TransMatrix
    ∇

    ∇ Delete w;sel
      :Access Public
 ⍝ Delete transaction with id <w> (and related splits/transfers)
      sel←(⌊|0⌷[1]TransMatrix)≠⌊|w
      TransMatrix←sel⌿TransMatrix
    ∇

   ∇ z←Detail w
      :Access Public
      z←,(w=0⌷[1]TransMatrix)⌿TransMatrix
    ∇

:EndClass
```

Sometimes it's simplest for the application code to go straight to the matrix, sometimes those little functions like AccountBalance make the application code clearer. At this time it's not really clear whether to continue accessing the matrix, or whether to write little method functions for everything – is the APLer clinging to the lamppost of life support? The only way to resolve this dilemma seems to be “write more code and it'll become clear”.

Notice that while our earlier “data class instances” were variables containing namespaces, this time we have a namespace containing variables.

Storing Class Data to File

Clearly we need to keep application data in some sort of file, and a component file seems adequate (we can get data in and out of DogMoney using standard interface file formats – which aren't adequate for operational use, and there's no need to share the working files with other applications).

Unfortunately...

```
     'c:\dick\temp\oo.dycf'⎕FCREATE 22
      payees ⎕fappend 22
DOMAIN ERROR
      payees ⎕FAPPEND 22
     ^
```

So we'll have to convert our data into a form that will go into a file...

```
   mf←⎕NEW #.UTIL.DogComponentFile(filename'shared')
   [ ...]
   mf.Replace 12(#.DATA.Types.(Id Name))
   mf.Replace 13(#.DATA.Categories.(Id Name Tax))
   mf.Replace 15(#.DATA.Payees.(Id Name)
   [...]
```

And read them back again...

```
   #.DATA.Payees←⍬
   payees←⊃mf.Read 15
   :For payee :In payees
      #.DATA.Payees,←⎕NEW #.CLASSES.PAYEE(1,payee)
   :EndFor
```

Which explains the case statement on our `PAYEE` definition...

And advice from bitter experience – avoid any temptation to store `⎕OR`s on file...

Scripts Without SALT...

DogMoney-specific code is held in the workspace – change the code, change the workspace.

But there's a lot that's common to other applications, and (in development mode) this code is saved as namespace scripts and read/fixed at startup (any changes to this code are detected when the application closes down and may be saved at this time).

SALT could have been used, but isn't. The core code for reading namespace scripts from file is straightforward...

```
   ∇ DogBoot w;⎕IO;⎕ML;tie
⍝ Boot DogLoader from file
 ⎕IO ⎕ML←0 3
 tie←w ⎕NTIE 0
 ⎕FIX{(~ω∊⎕UCS 13)⊂ω}1↓⎕NREAD ¯1 160(⎕NSIZE tie)0
 ⎕NUNTIE tie
 ⍝ -------- To write namespace script to file... -------------------
 ⍝    'c:\dick\mycode\dyalog12\tools\dogloader.dogalog'⎕NCREATE ¯1
 ⍝     ¯1 ¯2 ⎕nappend ¯1 83
 ⍝     (∊(⎕SRC #.DogLoader),¨⎕UCS 13)⎕NAPPEND ¯1 160
 ⍝     ⎕nuntie ¯1
 ⍝ ------------------------------------------------------------------
   ∇
```

All else is embellishment...

When not in development mode the code is locked and loaded in the workspace (could be made into an executable...).

So What Have We Found Out?

Even making allowances for misunderstandings of concepts and syntax, Dyalog's object-oriented extensions have broadened the expressive power of APL.

- `⎕NEW` and dot-syntax have expanded the syntax of APL in a useful and consistent way.
- Aside from a few remaining issues `⎕WC` (and friends) are effectively obsolete.
- A little effort defining your own GUI classes reaps rewards by reducing application-coding tedium.
- The spirit of APL is very much present when we take this syntax into the realms of application data.

And above all...

Even elderly APLers can do this stuff...

Further Notes:

- Some of the utility code within DogMoney has been plundered from Dyalog's distributed product (mostly DFNS) – I hope that this is adequately acknowledged.
- DogMoney code may be used for personal study and non-profit purposes only (no military, under any circumstances)
- The DogMoney application may be used as it stands, if it suits, but may not be distributed (either in part or in whole) to third parties without the express permission of the author.
- There are known (and unknown) errors in the DogMoney application.
- No responsibility is assumed for errors within either this paper or the DogMoney application, any use is entirely at the user's own risk.
- `http://www.dogon.myzen.co.uk/` contains a heap of related (and archival) writing.
- Not one word of this document may be reprinted or republished in any form without the express permission of the author.
- This document was produced using OpenOffice.org 2.4, which seemed determined to mangle “del” characters in an unpredictable way, despite all efforts by my devoted proofing staff it keeps changing them to something stupid every chance it gets...