

# The representation of knowledge

## Investigations leading to a simple computing concept

Something useful happens if you think of classes (in the object oriented sense) as tables (in the database sense). Something interesting happens if you allow classes to develop in response to the data they contain. Something clever happens if sets of data automatically understand the capabilities of their current members. Putting these novelties together is surprisingly easy from a programming perspective; leading to fascinating possibilities and real-world conundrums.



### **Peter Fox**

Peter Fox is an independent computer systems consultant living in Essex, England.

July 2008

More papers and contact details at <http://vulpeculox.net>

# The representation of knowledge

## Part 1 - Sketches

### Summary

This paper looks at a very simple to grasp but magically flexible programming system aimed at the storage and manipulation of data in such a way that the 'classes' of data items are enriched and dynamic, allowing operations to be carried out on what the data actually is rather than a rigidly pre-defined specification. It sounds very new-age but really it's a fusion of OO, relational database and simpler data definition schemes put together so that the system adapts and builds cross references.

The first part is a preliminary step-by-step exploration of how we can declare data units and add a bit of object orientation.

The second part sets out to clarify through discussion and experiment a definite programming language and system and highlight the important side effects.

The third part is a conclusion. Readers who want to jump to the results should start with the conclusion.

### Introduction

This is an exploratory paper looking at how we might develop a computer system that is practical from the point of view of an everyday programmer and which supports the development of meta-data in a way that is useful.

Throughout the discussion ideas evolve and problems surface. The value to the reader is as an introduction to the slippery emulsion that results when combining the oil of data with the water of meta-data. The more thoughts that are provoked the better as my objective in this part is more to define the issues and 'what do we really want to do' than specify answers.

Readers should have some rough idea of Object Oriented Programming and some passing familiarity with the sorts of style used in common programming languages. Sophisticates may blanch at some of my coarse code which is purely for illustration, but might also recognise important gaps which their background makes them qualified to fill-in. Good - That's what developing science and technology is all about.

## Basic elements

*Nodes* exist with a unique identity. Every node is an *Instance* of some *Class*. So for example the two Nodes *Asia* and *Europe* are instances, ie actual examples of the class *Continent*. All nodes have at least one Class which abstracts what sort of thing they represent. Person, Employee etc. (Here I'll try to use "Node" rather than "object" but the two terms are wholly interchangeable.) Nodes exist independently of all other nodes but depend on their class to make form out of raw data.

A Class provides the structure used by its members

- *Data structure* to organise internal data
- *Methods* to carry out actions
- *Interface* to interact within the system

A Node will typically have some *Property values* for example 'Height=10', Colour=Black etc. ('Colour' is a *Property* while 'Black' is a *Property Value*.) The Properties are conferred on the node by its class. For example all 'games' might have a 'rules' property while a specific game will have a specific property value for it.

Classes may be sub-classed. Class properties are inherited by sub-classes. For example if the class 'Person' has a 'name' property then so will the sub-class 'Employee'.

- Nodes describe data
- Classes define meta-data. That is data about data, or describing data.

Nodes may be associated with a Property as its Property Value. For example *Mary.OldestChild = Susan* where Mary and Susan are Nodes. (Something to watch out for: Node-Dot-something is a common way of indicating components of an object. Here "Mary" is a Node, that is a real thing - an instance, while "OldestChild" is a property of the class that Mary belongs to, that is meta-data. Humans are good at dealing with this but as we'll be ferreting around in this zone it is worth getting a clear understanding so we don't get tripped up when 'dots' become more complicated things.)

A Node may have *Relations*. A relation is a many-to-one connection between two nodes. These are specified as part of a class. For example the class Person may allow the Relationships 'ParentOf(Person)' and 'ChildOf(Person)' which allows instances of Person to form parent-child relations.<sup>1</sup> What should be made clear is that 'every plug needs the appropriate socket', relationships are double ended with specific connection rules. For example a Person node can't be added to a Continent node as if it was a Country.

There may be rules that limit the applicability of relationships between nodes of various classes. These might be complex and not be fathomable by the nodes themselves. For example, limited seating capacity on an aeroplane can only be 'understood' by looking at the whole plane not individual seat bookings.

A node may express properties it hasn't inherited from its ancestral class(es) by

(a) Causing a sub-class for it to be created in its own right

---

<sup>1</sup> Please ignore the specifics of syntax - this will evolve.

- (b) Joining a "Set" of 'things with property X' (specifically property X)
- (c) Multiply inheriting from a class other than it's 'parent class'.
- (d) Causing the property to be added to the properties roster of the class or ancestral classes.

A *Set* is a collection of nodes. *A unique class is always associated with a set.* Pawn and King might be members of the Set of Chess pieces and thus have all the properties of the 'Chesspiece' class. It is possible to have an empty set with an abstract class definition.

Another way of looking at a set is to say that the members all have at least one thing in common. This idea will become quite important later.<sup>2</sup>

## A convenient language

It's going to be really handy if we can use a 'programming language' to enforce a bit more precision to our thinking and to explore what sort of things we might want a computer to do for us.

Classes are prefixed by a  $\phi$  character and shown in upper case for clarity.

eg.  $\phi$ PERSON

Properties are indicated by a period prefix.

eg.  $\phi$ PERSON.Name

Relations are indicated by a colon prefix.

eg.  $\phi$ PERSON:Parents

Sets are indicated by a '\$' prefix.

eg. \$CHESSPIECES

Ghost classes, the classes associated with sets have the same name as the set but prefixed by  $\phi\phi$ . This is introduced in Part 2.

eg  $\phi\phi$ CHESSPIECES

Methods and Rules, which I'll always call *functions* are prefixed by a \*

eg  $\phi$ PERSON\*NumberOfChildren

Computer-style definition syntax is used for statements with result followed by method and I'm using the common '//' style of commenting and braces to bracket blocks of code.

Hint: One dot (property) is one item. Two dots (relation) allows more than one.

---

<sup>2</sup> There's no set theory here but a rough idea of where sets meet logic might be useful background.

## Exploring

First steps in putting together elements just to see what happens. Here I'm concerned with clarity for purposes of discussion of ideas rather than purity of syntax. Some elements used here will be superceded in later discussions.

### Example 1

Statement: `jim := New(φPERSON);`

Meaning: Create a node and give it the label "jim" for identification purposes. This will be of class PERSON.

Discussion: • This means the system can identify 'jim', and we can get the system to find 'jim' by some magic if we need to. But we could have an anonymous node created along these lines:

`mary*AddChild(New(φPERSON));`

- If the class PERSON doesn't exist then create it. What was that? Create a class even though we don't know anything whatsoever about it - and not only that but allow it to be used to create an instance! Yes, that's the idea. For example if someone tells you that "A platypus is not a mammal or a marsupial but a monotreme." then if you don't know what a monotreme is you still have that relationship between the instance and the class for elaboration later. Creating a system that absorbs meta-data is part of the plan.
- If a node identified as 'jim' already exists then...? This is an interesting subject because the answer is not always 'overwrite' as it would be in a standard programming language. Probably, but this needs research because it could lead to horrible mistakes. We are probably happy to let 'jim the φDOG' and 'jim the φPERSON' exist as entirely separate entities.
- jim is an instance that automatically belongs to the "set of PERSON things". Every class will have a set of things that 'are it'. Here the set \$PERSON gets another member. (This isn't too strange if you think of jim as a 'person database record in the person table'.)

**IMPORTANT**

**IMPORTANT**

### Example 2

Statement `φCHILD := SubClass(φPERSON);`

Meaning Create a class called CHILD which is a sub-class of PERSON

Discussion

- If Class PERSON doesn't exist then create it.
- If a class called CHILD already exists then ...? This is probably an error, the reason being that you can't have two different ideas in one shell - combining 'a taste' and an 'outboard motor' just doesn't work. However there may be distinct realms of knowledge where the names we give to things may be the same. "Bat" is a stick for playing cricket, a flying mammal and an action. We're almost certainly going to need to split our meta-data knowledge-space up into realms so this is something to bear in mind.

### Example 3

Statement `ϕCHILD := SubClass(ϕPERSON+$HASTOYS);`

Meaning: Create a class called CHILD which is a sub-class of PERSON and also a member of the set \$HASTOYS.

- Discussion
- This is quite a sophisticated operation illustrating multiple inheritance. Note that \$HASTOYS is a Set. (Sets are essentially collections of nodes with their class being developed 'behind the scenes') Suppose that ϕPERSON class exists and the set \$HASTOYS exists, then we're saying that the class CHILD will inherit directly from PERSON as in the previous example and also from the ϕϕHASTOYS class, ie whatever class underlies the \$HASTOYS set. This is dynamic meta-information.
  - Suppose instead the \$HASTOYS set doesn't exist at all, then **the system will create it** and the class ϕϕHASTOYS. This class won't have any properties at this stage.
  - We haven't created any nodes so nothing has been added to \$HASTOYS. This can only happen if we have a node as might now happen with the following statement:  
`Jim := New(ϕCHILD);`  
Now \$PERSON, \$CHILD and \$HASTOYS all have one more member than before.
  - Later we'll look at 'Labels' which might be a more specialised way of joining sub-classes with sets.
  - This is a very powerful scheme. In everyday life we're always being told about things that are 'like X with Y characteristics as well'.

### Example 4

Statement `ϕCHILD := SubClass(ϕPERSON);`

`AddProperty(ϕCHILD,Parent,ϕPERSON);`

Meaning We are defining the class ϕCHILD as a sub-class of ϕPERSON and adding a property (properties start with a period) and saying what type the property will take.

- Discussion
- This looks more like the functions we'd call in a kernel API rather than an OO high level language. However for our purposes here we can clearly see that we want to be able to develop classes after cloning them from their ancestors.

### Example 5a

Statement `Jim.Parent := Mary; // Note: Jim is a ϕCHILD object`

Meaning Jim is a node with a .Parent property (presumably as a result of being a ϕCHILD object). We 'set this to' the node (object) Mary.

- Discussion
- In the class definition for whatever object defines the .Parent property (possibly ϕCHILD) we will have declared what sort of thing can be assigned to it as a property value. This has all sorts of interesting implications:
  - Firstly, if we allow plain strings, numbers and primitive types then the data in the property value can only be a bit of 'leaf data' with no other contextual meaning. Suppose our definition is

`ϕPERSON.PhoneNumber = String;`

then valid Property Values could be

`Jim.PhoneNumber := "10334 545451";`

or `Jim.PhoneNumber := "Same as Mary";`

This sort of thing can lead to all sorts of conveniences:

```
Jim.Age := 6; // Age is a number
```

```
Tom.Age := 8;
```

```
oldest := Extract({Jim, Tom}, .Age, Max());
```

Which we can do if we have ready-defined numbers able to be operated on in a way to let us find the maximum.

- Secondly if `.Parent` isn't yet defined there are various possibilities.
  - (a) Refuse 'on the fly' creation of properties. This is a bit mean but is a useful fall-back position to prevent a 'wild tag explosion'.
  - (b) Ask for clarification for which class should acquire this property. It is easy to see that in a simple universe with `ϕCHILD` (being a sub-class of `ϕPERSON`) we'd want the new `.Parent` property to belong to `ϕCHILD`. [If `ϕPERSON` gets a `.Parent` property then it becomes in all respects identical to `ϕCHILD` - What happens then...?]
  - (c) Create a Sub-Class of the object specifically with this property. Suppose Jim was a node of `ϕPERSON` and we hadn't yet got a `ϕCHILD` class then it would be very easy for us to decide to create `ϕCHILD` with the defining characteristic that it has a `.Parent` property.
  - (d) Create a set, which automatically creates the class behind it, and make this node a member of the set (ie inherit the set's class).
- Thirdly note that if the property isn't yet defined we can define the type of property value that is acceptable by example. Interestingly there could be a number of statements with different types being used as examples. We might use this to modify the allowable type for the property value. One way would be to adjust according to the 'most recent common ancestor class'. This would need to be done 'intelligently' with similar alternatives as for 'Secondly'.
- Fourthly : If Mary is a node then there's a reflexive relation being set up such that "Mary (a `ϕPERSON`) has 'a property' of appearing as a value for a `ϕCHILD`'s `.Parent` property. In plain English we'd say that "Mary is a parent". This is quite important. If it is valid to assign a node as a property value then the class of that node must 'support somehow' a boolean property of "This appears as value for `ϕSOMETHING.SomeProperty`". We could go even further and define a `.Child` property for `ϕPERSON` and explicitly establish the value with:  
`Mary.Child := Jim;` If we're not careful we can get too complicated here  
- not least because we're really defining relations rather than properties.  
- More later.
- Fifthly : Once the type system has been established we can validate our assignment. Trying to assign the wrong class of thing ought to alert us to some mistake. Either the mistake is simply that we're trying to put apples into a pears slot or the way we've set up our knowledge representation is somehow flawed.

### Example 5b

Statement `Jim.Parent := Mary; // 1 only`

`Jim.Parent += Fred; // += Try to append another parent !`

Discussion Jim has got two parents. When assigning Mary as the property value we don't really have a problem, but when it comes to Fred we're trying to put a quart into a pint pot!

- We often want to add multiple property values. This looks like a Set, with the important proviso that this is private to Jim in this case. If we play our cards right we will be able to ask questions like "Which of Jim's parents is his mum?"
- If we treat this as a set then we can list the elements and test for membership and tell what the class requirements of the members is...
- Which leads us to the realisation that this is not really an X=Y type of property and more of a Relation where we're interested in the way our nodes are structured.
- The question is: Do we force  $\phi$ CHILD-dot-Parent to be  $\phi$ CHILD-colon-Parent as soon as it hits the first node assignment as a property value, or never or what?

What we've seen is an extremely powerful paradigm which can build sets and Relations according to simple cues, just as humans do. For the time being we can assume that there are useful operations we can define over sets in general, refined by the class (whether specific or virtual) behind the set. For example the largest or a random sequence, or the most 'recent' common ancestor class of all nodes.

### Example 6

Statement `$Children := {Eve,Tom,Jim,Sue}; // Four children in a set`  
`$Children += {Geoffrey}; // Add another`  
`$Girls := $Children*IsFemale; // Operate to obtain subset`

Meaning We create a set of four nodes which previously have been given the Eve, Tom, Jim and Sue labels. Presumably these nodes are  $\phi$ CHILD. Then we use a syntax that's familiar to programmers to add another to the set. Finally we apply the `*IsFemale` function (specified in some super-class of  $\phi$ CHILD) over the members of the set to extract the girls.

Discussion That looks nice and simple.

- We can use the methods we've already discussed when building a set. It may happen that all the nodes we're adding into the set have exactly the same class. This means there is an equivalence between  $\phi$ CHILD and  $\phi\phi$ Children. (The ghost class behind the set `$Children`.)
- We've also acquired an interesting possibility for deduction obtained by collapsing knowledge to leave just the differences between  $\phi\phi$ Girls and  $\phi\phi$ Children (and possibly  $\phi$ CHILD). We can look at
  - (a) Data : Which nodes are/not in the sets. Normal set operations.
  - (b) Meta-data : What the 'differences in common' are. Comparing the property values to conclude `$Girls` is a subset of `$Children` for a given reason.
- I've introduced a function `*IsFemale`. Perhaps we should discuss this a bit more: `*-`functions are methods defined in a class (or ancestor) to operate on nodes of that class. Suppose in  $\phi$ PERSON we had a property `.Gender` with enumerated values "M" or "F" or "?". Firstly  $\phi$ CHILD would inherit this property. Secondly we could define a function which examines this property.

```
ϕPERSON*IsFemale := {Gender == "F"};
```

If sub-classes have `.Gender` and the type of `.Gender` in these remains 'enumerated string' then there's no reason why they can't inherit the function.

- In the syntax of the example code we didn't apply the function directly to `ϕCHILD` or `ϕPERSON` nodes though. We applied it by implication to all the members of the set. In effect "For each Node in the set `$Children`, apply the `*IsFemale` function and collect any where the result is true." This is of course what we do all the time with database queries.

### Example 7

Statement `Jim:Children := { Simon, Sally };`

Meaning Jim has a set of nodes (dubbed a Relation) called `:Children` with two specified members.

- Discussion
- We already know about Sets so this isn't a shock.
  - The options for adding Relations are the same as for adding properties as we discussed above. Importantly we're dealing with the class of the node called Jim for the structure of the information "Things like Jim can have a relation called Children which appears to be populated with `ϕCHILD` nodes".
  - By populating Jim's `$Children` set (ie. Relation) we've established something about it that makes it different from say  
`Jim:Pets := { Rover, Tiddles };`  
where Rover and Tiddles are not `ϕCHILD` nodes but `ϕPET` nodes...
  - ... But it isn't hard to see how to compare Jim's pets and Jim's children to look for 'differences in common' ... for example `.name`, `.age` and `.gender`. We're very close to being able to ask questions such as "what are the names and ages of Jim's household?" leaving the system to worry about the exact meta-data.

### Example 8

Statement `Jim:Children := { Simon, Sally };`

```
Jim:Children += {Rover, Tiddles }; // Surely a problem here!
```

Meaning Adding the pets to the set of Jim's Children.

- Discussion
- We do want to allow some flexibility in being able to mix classes in a set but perhaps we've gone too far here. Let's analyse this a bit more.

Suppose we've introduced the Relation `:Possessions` to `ϕPERSON` by the use of a concrete example such as

```
Jim:Possessions := {Car, House, Lawnmower, MobilePhone};
```

where all the objects in the set listed on the right are `ϕOBJECT` or sub-classes of `ϕOBJECT`. It is not beyond the wit of a programmer to deduce the 'rule' that nodes added to `ϕPERSON:Possessions` must derive from `ϕOBJECT` as being the 'most-recent-common-ancestor' of all the nodes added so far. Notice that this 'rule' is a guess deduced from a very small population of real instances.

Now if we try adding something that isn't an `ϕOBJECT` derivative what should we do?

```
Jim:Possessions += Health; //OK?;
```

Clearly we need to have options because there will be cases where broadening the class constraints is desirable as in:

```
SSSaucySue:Cargo := {Coal, Stone}; // φMINERALS
SSSaucySue:Cargo += {Copper, Aluminium}; // φMETALS
SSSaucySue:Cargo += {Vacuum cleaners}; //φAPPLIANCES
```

Another interesting question is: If we remove items from the set do we re-apply a stricter Class interpretation?

One of the answers might be to endow classes with the (boolean) property "CanBeAMemberOfSomeSet". Notice how this is a class property so that all the options we've discussed about applying properties apply. So for example we might want to create a sub-class specifically.

## The scope problem

Suppose in my little world on my PC I've been manipulating data objects and developing the class structure behind it. To what extent should that enrichment of Class definitions and relations be exported to 'the wider world'? Perhaps I'm involved with teaching young people with learning difficulties, in which case I've probably got specialised data structures and classifications. How much, if anything is applicable to mainstream education? But wouldn't it be nice if I could start by using a mainstream education model? If I tweak it then does that improve the mainstream model or break it or confuse it or what? My hunch is that abstract packages might be made available in the same way as useful sub-routines in a conventional programming language.

Nevertheless because we can silently alter the meta-data by manipulating everyday data we need to be careful that within a single application we don't introduce weird rules and cruft. At some stage we need a 'what has changed' audit and perhaps a lock-down to manage drift in the meta-data. One possibility is to have snap-shot points for meta-data that can be rolled-back to.

## Trial run

Let's see what happens if we set up a small application. (Remember the syntax is not meant to be definitive.)

First attempt - Build the meta-data all in abstract

```
$DIARY := {φDAY}; // Collection of dates
```

- Create a Set called Diary.
- We can put dates into it.
- We haven't addressed issues of ordering and duplicates.

```
// -- some properties/relations --
```

```
φDAY.Date := DateTime; // DateTime being a built-in primitive
```

```
φDAY.Events := φEVENT;
```

- The system knows about φDAY from the preceding line so we don't have to do an explicit 'φDAY = NewClass' type of thing. In fact we may never need to explicitly declare an empty class if we can use context.
- There is a definite place for built-in types with proven functionality.

- It should be easy to see why `.Date` is a Property but `.Events` is a Set (Relation): The obvious one is that we could have more than one event on a day.
- When we defined `.Events` we indicated we wanted the set populated by `φEVENT` Nodes.

```
// -- some functions --
φDAY*Year := {return GetYear(.Date);};
φDAY*Month := {return GetMonthNum(.Date);};
φDAY*Day := {return GetDayOfMonthNum(.Date);};
φDAY*DayName := {return GetDayName(.Date);};
φDAY*New(Day,Month,Year){ .Date := BuildDate(Day,Month,Year);}
```

```
// Sub-class of DAY
```

```
φPUBLICHOLIDAY := φDAY;
φPUBLICHOLIDAY.Name := String; // Eg "Christmas"
φPUBLICHOLIDAY.PublicHoliday := True; // _Type_ of property is True
```

- `GetYear()` etc are assumed to be built-in functions for the `DateTime` type.
- Defining the `*`-functions for `φDAY` looks a bit tedious. A clever programming language would burrow into the class to find something applicable.
- The last line is quite interesting. Remember that we're dealing with meta-data, ie class structure, here not instances. Therefore when we say `.PublicHoliday := True` we're defining the *type* of the Property Value. Therefore 'True' must be a type in it's own right. The effect of this is to make `.PublicHoliday` a constant.
- `*New()` is a nod in the direction of constructors, destructors and accessor methods. More later.

```
// -- Events to attach to days --
```

```
φEVENT.Name := String;
φEVENT.Duration := φPERIOD;
φEVENT:Category := String; // or should this be := {Strings}; ?
φEVENT.About := String;
```

- This top-down approach seems to be working quite well so far.
- What we've done with `.Category` is make it into a set of Strings so for example "Dance" and "Music" could be both in there. This raises the *extremely* interesting question of constraints on a class-wide basis. (Can we add categories willy-nilly? - See Labels later.) 'Behind' the set/Relation is a Class of some sort and so far all we've done is constrained it to Strings. This is going to *need* addressing (along with some other loose set-issues from above).

```
// -- Durations --
```

```
φPERIOD.Start := DateTime;
φPERIOD.Finish := DateTime;
φPERIOD*OK := {return IsSet(.Start) and IsSet(.Finish) and (.Start <= .Finish);};
```

- We define a duration here with a start and finish points using (built-in) types for property definitions.
- However there is all sorts of possible grief waiting if we allow direct access to `.Start` and `.Finish`. What happens if one is set or the value of `.Start` is later than value of `.Finish`? This is where encapsulation, one of the strengths of Object Oriented programming, could be helpful and at the moment we seem to be ignoring that goodness. At least we've defined a validation function but that doesn't really cover all bases or avoid trouble in the first place.

- We would probably want to implement all sorts of period logic, and when we'd done that make it available for reuse. Clearly there's more detailed work needed here also.

// – Initialise diary –

ToDo := New(\$DIARY);

\*FillDates(ToDo,2007); // Cheat!

- There's a lot of suspicious hand-waving going on here. We've decided to execute some function which we must have defined elsewhere that will fill the diary with a year-full of dates from 1<sup>st</sup> Jan to 31<sup>st</sup> Dec.
- Once again this isn't very object oriented. ToDo is not a Class but a Set so we can't really define special operations for it. This would seem to indicate that we should be wrapping the collection of days in a Class so that we'd then be able to apply functions for it and so on.

// – establish a couple of public holidays –

pubhol -> ToDo\*Matches(New(ϕDAY(25,12,2007)); // Point to xmas day

SubClass(pubhol,PUBLICHOLIDAY); // Change class

pubhol.Name := "Christmas"

pubhol -> ToDo\*Matches(New(ϕDAY(6,4,2007)); // Point to GF

SubClass(pubhol,PUBLICHOLIDAY);

pubhol.Name := "Good Friday"

- pubhol *points to* the result of applying the \*Matches function over the members of the ToDo set. Well at least we can say it is readable, even if the underlying structure is grotty. Again we are limited by having ToDo as a bare Set.
- What do we mean by "matches"? How similar do say two cats have to be before they 'match'? We're going to need to be a whole lot smarter in this district.
- Pointers, as opposed to copies, are a common programming paradigm but we've got to be careful if mixing 'actuals' and 'pointers'.
- We've forced a sub-class on these two ϕDAYs. This looks like a good idea here but we're going to need rules and regulation.

## Another go

All-in-all so far what we've done is very messy and has a tangle of loose ends. This doesn't feel right. Lets try setting up the diary another way. Again I'll make up some code and then pick out the bones.

ϕDAY\*New( D, M, Y ) := { .Date := DateTime.Create( D, M, Y ) ; }; // Constructor

- This could be translated in English as "There's a new class called DAY and we've been given a constructor for it (ie how to create nodes) requiring three arguments of some sort... ...DAY must have a .Date parameter (create it) and we can tell what type it is from the (built-in) function that follows... We can also tell from this function what D, M and Y are allowed to be, so we can back-define D, M and Y in the \*New constructor for DAY". There are quite a lot of deductions about the meta-data in this short statement. In my opinion, being able to work out more than the obvious is exactly the sort of talent we should be looking for in a computer system.
- We haven't thought about what happens if invalid D, M, or Y arguments are supplied. This is a tricky question because, on the one hand it seems better to have a node flagged as defective so that other information is not lost, but on the other hand it

could be dangerous to have badly formed data floating around, and it is better from a policy point of view to strangle it at birth when we're more likely to have the contextual information relevant to the error. We'll try an example in a moment and follow through later.

```
ϕCALENDAR := SubClass(ϕORDEREDLIST,ϕDAY.Date);
```

- One of the issues we identified above was not having a specific class on which we'll be able to hang specific methods for the set of DAYS. Here we're creating a sub-class of some hypothetical pre-existing class ϕORDEREDLIST which for sake of illustration is expecting to be given a property that can be used to do the ordering.
- We might reasonably assume that ϕORDEREDLIST will have methods for finding the 'lowest' and 'next' and so on.
- One of the issues raised above was how do we implement 'equals' between objects. Let's assume here that as we've got a property for ordering that this will be the touchstone. ie All the things we need are provided by this handy superclass. This is what object oriented programming is all about; where we're being a bit clever is tweaking the meta-data without anything much being explicitly said.

```
ϕDAY.Events := ϕEVENT;
```

```
ϕDAY.Date := DateTime;
```

- We add an Events relation to ϕDAY and tell the system for the first time about a class called ϕEVENT. At the moment the system has no idea whatsoever about ϕEVENT other than we'll be using it.
- The next line is technically redundant as the system was able to deduce the type of .Date from the very first line of the code where the result of the DateTime.Create() function told us this information. However as careful programmers or diligent knowledge engineers taking a belt and braces approach this seems reasonable. Now what happens if this causes a conflict? (Suppose for example that when the system deduced the type of .Date above it decided that it was a Timestamp instead of a DateTime and now we're telling it something different.) We are going to need some way of deciding in the first place what we mean by "conflict" and then provide some appropriate scheme for error handling.

```
ϕPUBLICHOLIDAY := ϕDAY;
```

```
ϕPUBLICHOLIDAY.Name := String;
```

- Essentially we're sub-classing ϕDAY. There is an important semantic point we need to clarify because there's a crack which we could fall through here. Are we *really* sub-classing or simply cloning. If we're sub-classing then ϕPUBLICHOLIDAY will and *continue to* inherit everything from ϕDAY, so that if in a while we alter ϕDAY the change will be passed through to ϕPUBLICHOLIDAY. There is an alternative (which at the moment I don't see as being very useful) of taking a snapshot of ϕDAY then leaving both classes to develop entirely separately.

```
// – Flesh-out EVENT –
```

```
ϕEVENT*New( Start, Durn, Text ) := {.Start := Start; .Duration := Durn; .Desc := Text};
```

```
ϕEVENT.Start := DateTime;
```

```
ϕEVENT.Duration := Integer; // minutes
```

```
ϕEVENT.Desc := String;
```

- The first line is a three argument constructor.

- But what about argument types? These are 'hanging' until defined in the next lines. As we're only setting up the meta-data here we can afford to fill in the gaps in whatever order is convenient. ...
- ...But we are going to need some way of at least warning the user before use that there are ambiguities left hanging. This points to the need for a 'Meta-data monitor' which lives in the background as a spy on the meta-data model.
- Once again there are issues with defective arguments which need to be dealt with by some error handling.

```
Diary := φCALENDAR;
Diary*Add(φDAY*New(1 to 31, 1 to 12, 2007));
pubhol -> Diary*Match(φDAY(25, 12, 2007));
pubhol.Name := "Christmas"; // Not so fast!
```

- Some real data - A diary for 2007.
- Wow! The second line is a compact way to fill the Diary with dates. Possibly it is too concise for it's own good but the idea of being able to provide ranges or sets of arguments where traditionally only one would be expected or allowed is very appealing. We'll often be working with sets so being able to automatically iterate them into a function could be very handy.
- What happens when trying to create φDAY\*New(31, 2, 2007) ? As it stands this would be accepted because we haven't got any range checking in the constructor.... Ok then - What would we do if we did have range checking?
- The third line sets a variable pointer 'pubhol' to point to a node. This is a φDAY node and φDAY nodes won't have a .Name property so the last line will either fail or silently add a .Name property *to this particular node*. This is the creepy side of letting the meta-data be adapted by usage.
  - A really clever system would work out that we've already got a class that's φDAY + .Name called φPUBLICHOLIDAY and use that.
  - A not so clever system would invent a new sub-class of φDAY. This is still what we want, enriching data objects is precisely what we're trying to achieve.
  - A grumpy or watchful system would at least warn us, and perhaps prevent us, if we're trying to extend the properties of a node.

If we're not happy with the system guessing like this we could explicitly convert the node pointed to by pubhol into a φPUBLICHOLIDAY. Note that this messing with the actual node not copying as in a traditional cast operation.

## Discussion

The main points to arise from these two attempts at a programming language are:

- 1 Meta-data can be evolved from manipulating instances of real data. Amongst other things this means that playing with data can alter the meta-data model. This is both a benefit and a potential worry.
- 2 There is a clear distinction between data and meta-data. This means we need to monitor the state of the meta-data at the same time as, separately, managing the data.
- 3 Meta-data is pretty much equivalent to classical OOP. However in our system there's plenty of room for 'loose ends'. Whether we get tripped up by loose ends or find them handy for tying on bits as we go is a moot point.

- 4 The issue of dealing with errors continues to be ducked. One reason is that errors (or difficulties or ambiguities that need resolving) might affect the meta-data as well as the real data. For example suppose we set the .Date parameter of a `ϕDAY` node to the duplicate of an existing node in Diary. Assuming that somewhere there is logic that takes exception to this duplication of a primary key what is to be done? Is the intention correct and the logic wrong - if so we're going to need to sub-class `ϕCALENDAR` (which itself is a subclass of `ϕORDEREDLIST`) to allow duplicates. How can we do this safely? How could we reverse-out of a situation where a harassed user made the wrong choice? Who can we trust in the first place? Clearly this is a major issue requiring some policy decisions.
- 5 We came across some neat concise syntax and allowed all sorts of fudging. With a fudge-happy system that shouldn't be a problem, and if we're designing a system for convenience of use then why not let the system do the hard work of back-filling for us.

# The representation of knowledge

## Part 2 - Investigation

Here I'll look at some of the points raised by part 1 to see what sort of solutions and policies we could try.

### What's going on?

Our happy-go-lucky scheme of letting classes, properties and relations be created on the fly as the system deduces we need them is handy for letting us build meta-data without much of an overhead or restrictions. However there are many traps that could arise through sloppiness and confusion. For example suppose we've just been working with a class called  $\phi$ DAY but then mistakenly refer to a non-existent class called  $\phi$ DATE. The happy-go-lucky system silently creates a new class for us and we spend an hour staring at the code trying to find out why the damn thing won't work as expected. Can't we do better than that? Yes, but we need feedback from the meta-data system. I imagine this to be interspersed with the code a bit like an interactive teletype log of yesteryear:

```
 $\phi$ DAY:Events :=  $\phi$ EVENT;
  New class :       $\phi$ DAY
  New class :       $\phi$ EVENT
  New relation :    $\phi$ DAY:Events
    New set :      $DAY:Events
      Type :        $\phi$ EVENT
    New axiom :    ( $\phi\phi$ DAY:Events =  $\phi$ EVENT)
 $\phi$ DAY.DayOfWeek :=  $\phi$ DOW;
  New class :       $\phi$ DOW
  New property :    $\phi$ DAY.DayOfWeek
    Type :          $\phi$ DOW
 $\phi$ DOW.Dayname := String;
  New property :   Dayname
    Type :         String
 $\phi$ DOW.IsAWeekEnd := Boolean; // Deliberate mistake : D-zero-W
  New class :       $\phi$ DOW
  New property :    $\phi$ DOW.IsAWeekEnd
    type :         Boolean;
  *****
  Meta-data summary
   $\phi$ DOW
    .IsAWeekend : Boolean
   $\phi$ DAY
    .DayOfWeek :  $\phi$ DOW
    $Events :  $\phi$ EVENT
    ( $\phi\phi$ Events =  $\phi$ EVENT)
   $\phi$ DOW
    .DayName : String
   $\phi$ EVENT
    (No information)
```

## Something about sets and constraints

Suppose we want to be able to work out that "if today is Monday then tomorrow must be Tuesday". Humans are doing this sort of thing all the time. The infrastructure code would go something like this:

```
// How to create a Day Of Week class
φDOW*New( Name, WeekEnd ):= {.DayName:=Name;.IsAWeekend := Weekend;};
// These need arranging in a cyclic set
// Assume ENUMERATEDSET has methods for *Next etc and a relation :Elements
φDAYSOFWEEK := φENUMERATEDSET; // 'Sub-classing'
φDAYSOFWEEK.Cyclic := True;
φDAYSOFWEEK.Elements := φDOW;
// Make access easy
φDAYSOFWEEK.Key := φDOW.DayName;
// Add in some data
φDAYSOFWEEK*Add(φDOW*New("Monday",False));
φDAYSOFWEEK*Add(φDOW*New("Tuesday",False));
φDAYSOFWEEK*Add(φDOW*New("Wednesday",False));
φDAYSOFWEEK*Add(φDOW*New("Thursday",False));
φDAYSOFWEEK*Add(φDOW*New("Friday",False));
φDAYSOFWEEK*Add(φDOW*New("Saturday",True));
φDAYSOFWEEK*Add(φDOW*New("Sunday",True));
```

Now how are we going to force every DOW we use to be pulled from this pool? Not only is this an interesting question from the point of view of what code structure should we use but we also hit a grey zone that isn't quite data and isn't quite meta-data. In programming terms this is: "When we refer to a DOW instance are we looking at a unique instance or a reference to one of these seven instances?" Another way of recognising this is a significant problem is asking "Is every Wednesday the same Wednesday?".

The classical OOP approach is to make sure that the details of DOW are hidden inside DAYSOFWEEK and we *reference* these elements through DAYSOFWEEK. We've got a useful handle for doing this nestling in the sample code when we specified a .Key property. Now a key is just a reference - just what we want, and a fine way to make references explicit. Perhaps we need some notation to clarify what's going on in the code? Let's see how we might put these ideas together.

```
φDATE.DayOfWeek := φDAYSOFWEEK>Key;
or
φDATE.DayOfWeek := φDAYSOFWEEK>Index;
```

Where ">Key" says reference by the .Key property (which we had to define above) and ">Index" says reference by position in set (as in an array with a numerical index). So the first line reads: "The .DayOfWeek property of φDATE is a value which is used to access an element of φDAYSOFWEEK by the key property (ie .Name)."

Now what happens if we want to ask what days are weekends? Suppose we set up another key ... but we'll get more than one returned. That shouldn't be a bother if we're prepared for it.

```
φDAYSOFWEEK.IsAWeekend := φDOW.IsAWeekend;
then later
$DaysOff := φDAYSOFWEEK>IsAWeekend;
```

You may have noticed that we're straying into all sorts of set and dictionary territory here. The hypothetical class `φENUMERATEDSET` presumably wraps a Set and presumably a Set comes with useful tools. I'm rather keen to leave the exact implementation of these features to the reader - they're not difficult to imagine in general terms - and would only involve re-hashing collection classes from innumerable languages.

## Labels

It is quite easy to confuse "Monday" and Monday. The first is a string while the second is a word/label/tag/symbol which is automatically a 'pointer' to something that we want to refer to by that label. In our days of the week set we imagined that there was a dictionary look-up mechanism which takes a string argument and returns a node. That's all very well but it assumes everything has a name property we can use.

Data or meta-data or ...?
---------------------------------

It also means that we can't work out the node linkage at 'compile time'. Wouldn't it be nice to be able to say "The day that is Monday" rather than "The day (if it exists, if not raise an error etc.) that has the name property value of "Monday" in the list of days of the week."

The following code introduces '£' as the prefix for a label. We'll use this for referencing items within a set without having to give them explicit name properties.

```
φEVENT.Day := φDAYOFWEEK;  
φDAYOFWEEK.ShortName := String;  
φDAYOFWEEK.Abbreviation := String;  
φDAYOFWEEK.IsAWeekend := Boolean;  
φDAYOFWEEK*New( S, A, W ) := (.ShortName:=S; .Abbreviation:=A;.IsAWeekend := W; );  
// now fill set of DAYOFWEEK with labelled nodes  
$DAYOFWEEK£Monday := New("Mon","Mo",False);  
$DAYOFWEEK£Tuesday := New("Tue","Tu",False);  
// etc for all seven days  
  
// use this later  
SomeEvent.Day := £Tuesday;
```

At first sight we haven't gained much over something like

```
$DAYOFWEEK*Add(φDAYOFWEEK*New("Mon", "Mo", False),"Monday");  
then later  
SomeEvent.Day := $DAYOFWEEK*GetByKey("Monday");
```

because in truth there's a labelling/look-up mechanism whatever way we do it. However we have detached an object's *identity* from its *contents*. Some philosophers have made careers out of being confused by this. We don't need to get hung up about it so much but we can use another way of identifying objects to our advantage.

- By placing the label into a symbol table we can optimise lookup
- and validate code 'at compile time'
- and provide a layer of security by controlling who can do what to labelled nodes.

Also the syntax of "SomeEvent.Day := £Tuesday;" is clear and compact.

Something worth pointing out here is how we seamlessly switched between the class `ϕDAYOFWEEK` and the set of all nodes of this class `$DAYOFWEEK`. In the last line we're so cool that we can leave the system to work out what `fTuesday` is all on its own.

I will leave labels for now as they might confuse other aspects that I'm trying to highlight. However feel free to say "Hey! - Use labels" in some of what follows.

## Axioms

One aspect we haven't really nailed-down is how `ϕDOW` should be tied to `$DAYSOFWEEK`. We're surely going to need some way to distinguish between using a `ϕDOW` reference and fiddling with the meta-data associated with it. What would happen if we tried to add another day of the week? What about changing Thursday to a Weekend? At some stage we're going have to declare these matters fixed.

This applies to a lot of meta-data, not only in meta-instances but also functions. A simple example is where we want to define a method for "earlier of two dates". For *any given class* there can only be one method. This might cause problems when we make assumptions about the number of possible answers to a single question. 'Optimum' might mean different things or there might not be a sensible answer - the question might not even be applicable.

As we develop our meta-data model we may need to enhance our methods. One error situation is where our methods are not sufficient to deal with the enriched data. For example we might start building a `ϕDATE` class with the assumption that a date can be represented as a point on a timeline. So that "12<sup>th</sup> March 2007" is represented by (say) an integer of  $7 * (366 * 24 * 60 * 60)$  seconds +  $2 * (31 * 24 * 60 * 60)$  seconds +  $12 * (24 * 60 * 60)$  seconds. ie roughly the number of seconds since midnight on New years day 2000. Now try to answer the question "Is 7:15am on the 12<sup>th</sup> March 2007 'earlier' (or 'later') than 12<sup>th</sup> March 2007?" Our *logical model* is not capable of answering this question as a whole day is not a 'point' but a range. On the other hand our *implementation* may mis-represent our model and lead to unanticipated problems.

A simple rule of thumb we can use is that *functions are axiomatic*. Lets back-track a moment and look at that set of days of the week. How we look up a day of the week by index or key is really a function so we might say that it is axiomatic that the first (lowest index) day is "Monday". Useful though this insight might be, we're still left with the possibility of the data, which is practically meta-data, being modified. There's nothing for it, we'll have to provide a statement to say that a certain meta-instance (thin and slippery ice!) is now immutable.

<b>Important</b> Functions and constructors are axiomatic.
---

Constructors are functions and are therefore axiomatic. (Let's ignore the issue of multiple and dynamic constructors for now. This shouldn't affect axiomaticness, in fact we might usefully insist on it.)

## Breaking functions

Consider the following code

```
ϕCIRCLE*New(X, Y ,Diameter)={.OriginX:=X; .OriginY := Y; Radius := Diameter / 2};
```

```

Wheel1 := φCIRCLE*New(4,5); // Not enough arguments
Wheel2 := φCIRCLE*New(4,5,-6); // Silly Diameter
Wheel3 := φCIRCLE*New(4,Red,5); // Argument issue?

```

If we're letting the system build its meta-data on the fly then the first line is interpreted as:

```

New class φCIRCLE
New property φCIRCLE.OriginX   Type : unknown
New property φCIRCLE.OriginY   Type : unknown
New property φCIRCLE.Radius     Type : number
New function φCIRCLE*New        Arguments : Unknown, Unknown, Number

```

Which is fine and dandy. We're axiomatically (because \*New is a function and we're acting on our hunch that functions are axiomatic) defining what we need to know to create a circle.<sup>3</sup>

We could have done something *similar* as follows:

```

φCIRCLE.OriginX := Real;
φCIRCLE.OriginY := Real;
φCIRCLE.Diameter := Real;
// no constructor defined

```

Then created new wheel nodes property by property. This method allows us to create wheels without origins or radii which probably isn't a good thing.

Let's think about what happens when we try to create Wheel1 with only two arguments. The constructor function requires (axiomatically) another argument. A computer system would report something like

```
Error - Insufficient arguments in line 999
```

while we, with our more philosophical viewpoint might say : "I'd like to create a CIRCLE object for you but it's impossible to do that thing unless you give me the diameter - It's axiomatic that all circles must have a diameter". It's the same thing really. I have a feeling in my water that "axiomatic failure" is going to be an important category of error message.

When we create Wheel2 there is no problem reported because we've not done any validation and allowed a negative diameter to be set! How should we deal with this?

There are two basic philosophies and we could use both:

- Validate function arguments
- Validate properties (singly and possibly in combination)

Lets scribble down some code to see what this might look like

```

φCIRCLE*New(X, Y, Diameter):={
    Assert(Diameter >= 0);
    .OriginX :=X;
    .OriginY := Y;
    Radius := Diameter / 2;
};
φCIRCLE.OriginX := Number;
φCIRCLE.OriginY := Number;
φCIRCLE.Radius := Number; {
    Assert(.Radius >= 0);
};

```

---

<sup>3</sup> The .Radius parameter must be numeric because we have performed a numeric operation on the incoming Diameter argument. That also means the Diameter must be numeric.

This looks interesting. The first point to note is that the system can tell that X in the constructor has to be of type Number (or something that can be cast to a Number) by back-deduction from .OriginX being a Number and the assignment in the constructor function. Secondly we've introduced some magic sanity-check function Assert(). Many readers will have seen this construction in programming languages before. It's job is to sound the alarm when the condition inside the brackets fails.<sup>4</sup>

In theory if we have the assertion in the definition of the .Radius property then we don't need it in the constructor but it's a fairly well established principle of programming to try to spot problems at the earliest possible opportunity - for one thing we're likely to get more context out of the error message. (So far we haven't introduced any mechanism for preventing 'outside interference with internal matters' which many OO languages provide to reduce the likelihood of this happening.)

OK        Negative Radius not allowed

Better    Can't create a CIRCLE called Wheel2 with a negative Diameter

Which brings us to a point of utility that we can consider : Enriching the message provided when the assertion fails. Not only is this good for illuminating the problem if it should happen in real-time but anyone reading the code has it explained as well. So for example the first assertion above might be enhanced as:

```
Assert(Diameter >= 0 , "Diameter must be positive"); // Code watch warning: Is 0 'positive'?
```

Creating Wheel3 in our original code is acceptable because .OriginY is not specified as any particular type so we can get away with passing anything to .OriginY which means we can pass anything to argument Y in the constructor. (Note, in some cases this is exactly the behaviour we want with the meta-data adapting to instances, but obviously here it would cause a problem later.) This issue is resolved when we explicitly define the type as we did later, or have functions that use .OriginY which expect a number. In fact we're automatically invoking an assertion that the type of a property value is acceptable as a result of having specified the property type. If that's the case then we can say that when we explicitly specify types we're being axiomatic.

## Where's that error stem from?

When we tried to give a negative diameter to a circle we could reasonably understand that 'in real life you just don't have such things'. That is the meta-data used to model our world knowledge was correct and the instance data was at fault. But what about a diameter of zero? Is that 'allowed'? The assertion code above allows it but we don't know if that's really sensible. What happens if we're working out the pressure on the end-cap of a cylinder?  $Pressure = Force / Area$ , but a zero diameter will give us a zero area and a weird and meaningless divide-by-zero result. In this case both the data and the meta-data are at fault. There shouldn't be a zero diameter (instance) anyway, but neither should we be allowing it in our (meta-data) model.

Of course lots of these cases lurk unseen and unsuspected in programs (meta-data) but never get exposed because the data is well enough conditioned... until...

If we return for a moment to the first go at making wheels what would happen if we executed the following code?

---

<sup>4</sup> Although in most programming assert() is only switched on for debugging and development.

```
φCIRCLE*New(X, Y, Diameter):={.OriginX:=X; .OriginY := Y; Radius := Diameter / 2;};  
Wheel4 := φCIRCLE*New( GrizzlyBear, "Blue", 5);
```

After the first line we've got no idea what X, Y, .OriginX and .OriginY types should be. Then along comes an instance that we can deduce these things from! The second line is perfectly acceptable to a system that learns from its inputs. The thing about meta-data is that we're probably not in a position to say where or when it will be used. (Importantly, there are ways to manage this given good OOP practice - but here we'll follow the observation that once something 'gets written down' it's very difficult to unwrite it.)

Now what happens when some time, possibly days or months, later we try to create a φCIRCLE node the right way. Oh dear! X and .OriginX are now flagged as being of type φURSINE which conflicts with a plain number provided in good faith. Now the root of the problem lurks in the meta-data alone. We could be hard pushed to discover where this erroneous notion that .OriginX is a bear came from. Furthermore should we be expecting some poor user who is only trying to draw a pretty picture of a bicycle to wade through this mess? There's a philosophical point in here: Some human is responsible for, and one hopes knows about, the original meta-data and should be aware of the environment in which it is evolving - how do we identify and reach them?

The example given is pretty clear cut but where there is confusion between related sub-classes and dynamic Sets with dynamic members of varying classes this could be a very mysterious situation.

It's not too difficult to conclude that we need a meta-data audit trail. In one sense this is easy, just record where the meta-data changed and whether it was axiomatic or deduced. In another it is fiendishly difficult as the trail may lead to external systems, transient data, remote libraries, long-since removed source code and interactive events. On the positive side it is generally useful for a system that evolves to be able to explain how it arrived at the current position. In a full-featured class we're quite likely to have enough axiomatic and deduced-at-class-definition-time information to clarify property types, but there are bound to be more subtle effects that arise from creating classes on the fly as we often do each time we tweak the elements of a Set.

## Ghost classes

This was briefly touched on before but as it's such an important aspect of how sets of information develop their own characteristics is worth going into in a bit more detail. (A Set called \$Foo will have a Ghost class called φφFoo. Also we'll often refer to a set by name without the \$ prefix because it is a set by definition.)

A Set works on two levels - data and meta-data: A set of objects can be examined and manipulated if there are common principles that apply. For example we can count any mixed bag of objects.<sup>5</sup> If there is some metric that is common to all the objects then we can compare, sort, and obtain an average. If there is common function applicable to all the objects then we can apply that to all of them. Notice that to do anything more than

---

<sup>5</sup> Even this isn't strictly true. In the physical world one object is one object, but in byte-world we may have multiple pointers to the same thing: Tom and Dick are twins... How many parents do they have? Two or four?

counting we need access to the class (or built-in type). This is of course the meta-data level.

A ghost class comprises the (intersection of) aspects of all classes associated with all objects in the set. So for example

```
Figures := $SHAPES;  
Figures*Add(ϕTRIANGLE*New(SomeArrayOfPoints));  
Figures*Add(ϕSQUARE*New(AnotherArrayOfPoints));  
Figures*Add(ϕCIRCLE*New(APoint,6)); // radius of 6
```

All of the things we added had a class that knows (we assume) what \*Area is, so we can say that the ghost class ϕϕFigures has acquired that function. This is *reverse inheritance*. The classical OOP method would be to define an abstract class (or interface) with the abstract method \*Area then implement the function in each subclass. Notice that we've said nothing about \$SHAPES, ϕSHAPES or ϕϕSHAPES. Suppose we continued coding on from this example with

```
Drawings := $SHAPES;  
Drawings*Add(ϕTRIANGLE*New(MorePoints));  
Drawings*Add(ϕSQUARE*New(EvenMorePoints));
```

then ϕϕDrawings will be able to implement a \*HowManyStraightEdges function (acquired we assume from the methods of ϕTRIANGLE and ϕSQUARE) but ϕϕFigures won't be able to because it contain a circle which (for the purposes of illustration) doesn't implement that function. \$SHAPES will have 5 elements, (\$)Figures will have 3 and (\$)Drawings 2. Note that the methods here work much along the lines of interfaces, where all we're doing is discovering a function name/signature that can be implemented.

This mechanism, working out what all elements have in common, is important (but risky as we'll see) because the system is able to learn as it goes along and tell what operations and properties are appropriate to all elements of a set.

There is an important characteristic of sets and their objects that must be emphasised. *There is no need for nodes in a set to be derived from a common ancestor class.* For example:

```
Things*Add( LeaningTowerOfPisa );  
Things*Add( JimSmith );  
Oldest := Things*Max(*Age);
```

Here, so long as all the elements of the set Things support the function \*Age, we don't care about any other properties or functions. Isn't that practical and convenient!

**IMPORTANT**

### **The trap of latent meta-data modification**

We could easily be extracting elements from dynamically created sets (and therefore potentially dynamically changing Ghost classes) to suit our needs such as:

```
// End of month balance of payments  
Credits := Invoices*IsUnpaid; // Create set of outstanding invoices  
Debits := Bills*IsStillToPay;  
Balance := Credits*Sum(*Amount) - Debits*Sum(*ToPay);
```

[Line 1] Create a set called Credits being those where the elements of the set Invoices return True to the \*IsUnpaid function. [Line 2] Create a set called Debits where the elements of the set Bills return True to the \*IsStillToPay function. [Line 3] Return the overall credit in the variable Balance by summing the amounts in Credits and Debits.

There's nothing very remarkable in these lines. But what would happen if half way through the month by some mischance a `cGRIZZLYBEAR` node without a `*IsUnpaid` function is added to `Invoices`? If the ghost class behind `Invoices` is purely reactive then the set `Invoices` loses it's ability to implement the `*IsUnpaid` function across all its members and we're in a pickle until we can undo the mischief. So we're going to need the ability to insist, at least in part, what the minimum specification for nodes being added to a set should be. If these lines of code had been compiled into the system before any data was added then perhaps the fact that we call this function on all members of the set would be specification enough, but that's a big 'if' which can't be relied on. The bottom-line is that we need to be able to add axiomatic definitions to ghost classes early on in their 'lives'.

## Type

We've seen that knowing what sort of thing is suitable as a property value is a useful facility. We've also seen that collections need some type sensitivity if they're to be more than just a bag of miscellaneous objects.

We could have a stores system that worked on the basis of physically pointing at the things you want and asking for 'a handful of those', or a stores system that worked by typing-in a part number. Interestingly, in real life most general purpose practical stores systems have a combination where small consumable items are there for the taking without paperwork or parts numbers. Some stores don't have uniform manufactured parts - a wood yard might have everything from whole trees of various woods, shapes and sizes through to off-cuts. The reason for looking at these various types of store is to illustrate that in the real world it isn't always appropriate to impose strict typing - A wood yard can't have part numbers for every type of tree, bough, wood, beam and plank; some other descriptive scheme is required.

For a wood yard system we'd perhaps define a set `$STOCK` which automatically creates a ghost class `c$STOCK`. Now we can start putting items into `$STOCK`.

Add some sawn planks of type `cSAWNTIMBER`

Add some fence posts and some chippings as type `cGARDENCOMPONENT`

Add some tree trunks as `cSEASONINGTIMBER`

Fairly obviously we're going to need some common elements to these varied classes.

We have two possible methods to create these:

- 1 Explicitly create a super-class then derive these as sub-classes.
- 2 Allow the ghost class `c$STOCK` to learn for itself what aspects these classes have in common.

The first approach is traditional, axiomatic but rigid. The second approach is tempting, flexible but haphazard. In my view we should use a combination of both, to be used in the appropriate proportions. In fact having sets that can work out for themselves what's possible with their elements is a fundamental aspect of what I'm proposing.

We could start coding the above as

```
c$STOCK.Description := String;  
TimberYard := $STOCK;  
cSAWNTIMBER := SubClass(c$STOCK);  
cSAWNTIMBER*New(Desc,Wood,Type,Width,Height,Length){ .Description := Desc; . . . };
```

Which in narrative form is

[Line 1] The `ϕSTOCK` class (and hence anything in `$STOCK`) has a property called `.Description` of type `String`. [Line 2] `TimberYard` is set of `ϕSTOCK` items. [Line 3] A new class is defined called `SAWNTIMBER` which is a sub-class of `ϕSTOCK`. [Line 4] Here's the constructor for the `SAWNTIMBER` class where (amongst other things) we are given `Desc` as a string to assign to `.Description`.

I don't want to insist that all items in a set should derive from a common ancestor class. This is more important than it might at first appear. The reason is that if everything in a set has to be a sub-class of a given class then all objects and their classes must at some stage have had access to the 'master' class. This makes importing perfectly compliant objects from remote places impossible unless there's a definite standard which everyone uses (and which never changes). It's a bit like the international postal system: The sender buys a stamp locally of the sender's country and the various postal authorities sort out their differences in bulk. I don't have to come to your country to buy a stamp from your country's post office to be able to send a letter there. That would be silly - So is assuming that every object in a system is locally accessible and always available.

There is an answer that neatly retains the 'put it in the set if you like' flexibility while maintaining some 'lowest common denominator' constraints. If some *functions* are defined over the set as a whole these then are axiomatic (by definition - from a previous hunch) and so anything added to the set that doesn't support such and such a function will cause an exception. Lets see how that works:

```
ϕGROCERIES*Description{ return .Description; }
ϕGROCERIES*BarCode{ return .Barcode; }
ϕGROCERIES*Price{ return .Price; }
ShoppingBasket := $GROCERIES;
```

**IMPORTANT**

What we've done is insisted, axiomatically because they're functions, that any element of the set `$GROCERIES` will have a `.Description`, `.Barcode` and `.Price` properties. There's nothing anyone could complain about there is there? Is there? What happens if we add a handful of carrots to our shopping? Carrots only get weighed and priced at the checkout and don't have barcodes! (They wouldn't be `GROCERIES`-compliant.)

- The first thing the system can do is recognise the problem.
- What happens as the second thing is more debatable.

If there were no constraints on what could be added then `ϕϕShoppingBasket` would shrivel to the properties that all elements had in common. Say in this case just `.Description`. This is the 'lowest common denominator' effect in action. The meta-data monitoring system might log the change to meta-data but would otherwise be unperturbed.

But given that we have set up some constraints we now have three choices:

- 1 Disallow adding to the set "Hey that's not an X - you can't add that here"
- 2 Allowing it to be added 'under caution'.
- 3 Force the addition of properties and default methods to the node thus creating a 1-off sub-class.

In a practical supermarket situation we use method 2 and get the loose vegetables weighed and priced at the checkout. Method 3 is too tricky to apply generally but could be used given certain restrictions which will be discussed shortly.

From a computer programmer's point of view it is 'very naughty indeed' to break the rules. On the other hand when we allow for modification to meta-data this could be a

very useful clue that our assumptions about how the world works are defective. There may well be situations where we particularly want to examine the non-compliant components for where they're lacking, but still keep them together in the parent set. (Of course we've really split our set into two exclusive sets; and if we've tuned our programming language to support this we're not in any trouble.) Let's take submissions or reports of one form or another such as student essays, job applications, case histories or investigatory reports. Just about our first job is to see if they meet our basic standards when they're first submitted. If they don't then these might be the ones we have to examine first to get the problem fixed, use another method, or flatly reject.

If we're going to reject we ought to say why.

"Sorry your application for a Cataract operation was turned down because  
- no entitlement information was included on the form  
- no referring doctor's signature"

This is a very common clerical activity : Does this document meet all the requirements is a basic weeding process.

## Sets as processing tools

### Mutating objects

Similar processes can be used for sorting into priority, splitting into streams, allocating to people, further processing, appropriate indexing, waiting for matching information and eventual deletion. Let's consider a job application. It will

- (a) pass through a number of stages
- (b) have a number of decision points
- (c) be enriched as it progresses.

In code (a) and (b) might look something like:

```
$APPLICANTS*Add( Appn ) := {Assert( Appn is φOURJOBAPPFORM )};  
FirstLevel := $APPLICANTS;  
FirstLevel*Add( JimSmith );  
FirstLevel*Add( JaneBrown );  
// etc ...  
StrangeApplications <<- FirstLevel*Failed( Add );
```

Which we could read in narrative form as [Line 1] This is only for applications made on our custom job application form. [Line 2] Create a set for applications as they arrive and [lines 3+] add them. [Last line] Transfer into a new set any applications that failed at the Add() function. (The strange applications might easily contain the most interesting one. "You have all my details as I already work here, know the business and have an unblemished employment record")

(c) was briefly mentioned as the third possible option when discovering that an element doesn't have the necessary characteristics to 'fit' in a set. Let's consider those job applications again - When they first arrive they're a bit naked and we're going to have to add various scores, results of questions, suggested processing pathways and so on. But isn't it rather peculiar to modify the core structure of an object because it is lodged in a particular place... ..and what happens if the object moves on elsewhere, are the changes permanent? What we've got here is a practical need (enrichment) on the one hand with something that seems quite odd (fiddling with the original data) on the other.

To give a practical analogy: At the end of a manufacturing process a part might have an inspection stamp affixed to it. From an information point of view an uninspected part doesn't advertise the fact that it's uninspected (unless there is a place on the item that explicitly states "When inspected the inspector's stamp will go here - Until then it is uninspected"). That is, an absence is no information at all unless there's some axiomatic flag to say we haven't yet got the information. Or to put it another way, if the none of the pages of this paper were numbered how would you know if any were missing?

This discussion is pertinent to the (c) issue because we need to be clear in the first place whether we're :-

- 1 modifying the object's structure (adding something that wasn't originally there)
- 2 altering a parameter (putting an inspector's stamp in the place waiting for it)
- 3 wrapping it (putting it in a bag then labelling the bag).

These real-world examples are relatively easy to grasp but become slippery in the abstract. We need to understand this in the context of Ghost classes evolving as elements are added *and taken away* from sets.

(1) The proposed system deliberately makes no mention whatsoever that classes should have immutable structure. We have set out with the intention of allowing meta-data to be modified explicitly and as a result of interactions. This is going to get us into trouble if we start hacking with *some instances* of a class. Therefore we're going to need to have a separate class for which the nodes (instances) can be derived from nodes of another class.

```
StubForIndexing := φKEYWORDS*New( SomeDocument ); // Create abstract from document
```

(2) If we've built-in all the options we need in the first place then all we have to do is twiddle the appropriate parameters. There is a minor issue of how we set default values which has to be axiomatic.

(3) Wrapping again needs another class to be the wrapper. It's very simple, the original class becomes a property of the wrapper and additional features are added to the wrapper. One of the things about wrapping like this is that we could 'take the original object out of the wrapper' at a future stage.

```
φSTAGE3.BitsSoFar := φSTAGE2;  
φSTAGE3*New( Application ) := { .BitsSoFar := Application; };  
SeriousCandidate := φSTAGE3*New( PossibleCandidate ); // Passed qualifications check
```

What's the difference between (1) and (3)? In (3) we're tied to the original object whereas in (1) we've created a completely new object by analysing the properties of an instance. We can conveniently label these three approaches as follows:

- |                   |  |
|-------------------|--|
| (1) Metamorphosis | (Butterfly can't be turned back into caterpillar!) |
| (2) Enrichment    | (Only one object)                                  |
| (3) Wrapping      | (Dependency on original!)                          |

## **Mutating classes in the context of a set**

Let's look again at Ghost classes.

It may be becoming apparent that Ghost classes are not really any different from plain classes. However we'll continue to use this concept to emphasise a class that tends to derive its characteristics from nodes in a set membership.

Suppose I have a set containing instances of two simple classes.

◊RECTANGLE has .Width and .Length properties and an \*Area function

◊CIRCLE has a .Radius property and a \*Area function

The ghost class, being the *intersection* of the meta-data of the member classes, only has the \*Area function.

Now suppose we are going to work out a cost we'll need to add a .UnitMaterialCost property (or perhaps a function based on .Material) (to multiply by \*Area). ◊CIRCLES don't have any such baggage as costs and materials they're pure geometric figures. We could specify the necessary properties and functions as additional to the Ghost class.

Let's see if this would work by looking at some code.

```
Plates := $SHAPEDPIECES;
Plates*Add( Patch ); // an instance of ◊RECTANGLE
Plates*Add( Square ); // an instance of ◊RECTANGLE
Plates*Add( Cap ); // an instance of ◊CIRCLE
// Axiomatically add a material property and the way to price an item
// to the ghost class.
◊◊Plates.Material := ◊MATERIAL; // MATERIAL has a UnitCost property
◊◊Plates*Cost := { *Area * .Material.UnitCost; }; // define function 6
// Will the next line work?
Plates*Sort( *Cost ); // Cheapest first
```

We have added three geometric shapes (real data) to the set. Then we twiddle with the meta-data of the ghost class to add a material property with a type of ◊MATERIAL which we assume has a .UnitCost property. Unfortunately we can't sort using the \*Cost function until we've set the extra .Material *property value* for all of our pieces. So as it stands this scheme is unworkable as we don't have enough data.

Here is a possible answer: Deal with these axiomatic matters when the node is being added to the set. Of course this means we need to specify what the axiomatic matters are *before* accepting real data. In this example one axiom is that a \*Area function must be supplied by the object being added and the other (unit cost) needs to 'come from somewhere else'. Here's how:

```
Plates := $SHAPEDPIECES;
// Axiomatically add a material property and the way to price an item
// to the ghost class.
◊◊Plates.Material := ◊MATERIAL;
◊◊Plates*Cost := { *Area * .Material.UnitCost; }; //(last star is 'times')
// Now add some real data items
Plates*Add( Patch, Steel ); // an instance of ◊RECTANGLE plus material
Plates*Add( Square, Aluminium ); // an instance of ◊RECTANGLE plus material
Plates*Add( Cap, GlassFibre ); // an instance of ◊CIRCLE plus material
Plates*Add( Stick, Wood ); // will fail 'cos a stick won't have an *Area function
// The next line should work OK
Plates*Sort( *Cost ); // Cheapest first
```

(Assume that the second argument of \*Add() will get assigned to .Material)

---

<sup>6</sup> The second \* is used for multiplication as in Area times areal cost.

There are four axioms defined here:

- Ghost class `.Material` is a `ϕMATERIAL`
- `ϕMATERIAL` has a property `.UnitCost` (and must be a number because it is used for multiplication)
- Ghost class function `*Area` must exist (and return a number)
- Ghost class function `*Cost` must exist.

If these are axiomatic then they're essential aspects of our meta-data system which we need to enforce.

- When we try to add an instance of a class which doesn't have an `*Area` function we should get alerted to the fact. As there doesn't seem to be any other source of a definition for `*Area` we can use we know we're stuck.
- As `*Cost` is supplied by the Ghost class we're guaranteed (providing we've got the other axiomatic elements `*Area` and `.Material`) to be able to calculate it. So we're safe using it as a metric for sorting on.

What happens if we added an item which supplied its own `*Cost` function? In an ideal world perhaps we'd :

- (a) have satisfied the need for the function to exist for this item
- (b) use the item's class's function in preference to the ghost-class's version

and if we were really clever :

- (c) not need the `.Material` property if it was *only* needed for `*Cost`.

There is a tricky converse to this issue which is where we want to apply a different function (or property type) to an element. (This brings us back to the wrap/mutate issue discussed above). For example a practical sheet-cutting program might want to override the `*Area` or `*Cost` function. A circle might have to be cut out of a square sheet with the corners thrown away as scrap in which case the area *for cost purposes* is four times the `.Radius` squared not ' $\pi$ -r-squared'.

## Simple set operations

Before leaving sets we ought to quickly visit the more mundane aspects which will form an extremely important aspect of any implementation.

Firstly, the term 'Set' is used in the general sense of a collection rather than the 'no more than one of anything' sense. (An easy way to remember this is to say a "Chess set has 32 pieces" - not 6.)

As is to be expected, some of these operations relate to meta-data while others work with nodes of real data. We've seen how these can interact - for example adding depending on axioms.

Item manipulation operations such as adding, pointing-to, removing, copying. We've already seen some of these are more complex than at first sight. Also we might consider low-level item-specific operations such as testing for acceptability and issues associated with serialisation.

Counting and random enumeration are a very simple set-level operation that are independent of the classes of objects contained in the set. We can consider variants for each of these once we have access to their properties and functions.

- Testing for conformance to some standard. For example "Does this node have a .Description property and is it a String?"
- Testing for defective or incomplete nodes.
- Counting those matching a certain condition (data or meta-data)
- Ordering

Collective operations such as totals and averages are fairly simple to implement if we have access to suitable properties or functions.

Importantly we may want to return something much more complex than a simple statistic. For example suppose we had objects representing the rainfall each month for a year (`ϕYEARSRAIN.Month1 ... .Month12`, practically an array of 12 elements) then we might think it really handy to be able to say something like the following in order to get the average rainfall for each month totalled over many years. Here `LotsOfData` is a set of `ϕYEARSRAIN` objects.

```
AvAllYears := ϕYEARSRAIN*New;           // 12 slots just like the raw data
AvAllYears := LotsOfData*Average;       // fill the slots
MostAverageYear -> LotsOfData*Nearest( AvAllYears ); // Pattern-match heuristic!
```

What we're doing in the second line is analogous to totalling a range of cells in a spreadsheet and writing them along the bottom or looking at database records.

This might seem a bit ambitious but it shouldn't take too much working out what is required. Of course `*Nearest` is a bit of a joker in the pack, quite likely requiring some heuristics, but finding 'near' things is a very useful capability and needs at least a framework.

To go over some ground again; when we say "`SomeSet*SomeFunction`" we are happy for this to be interpreted as "For each element perform `*SomeFunction` on the element". (This syntax could conflict with operations that are meant to be applied to the set rather than the elements in it but we leave that as an implementation detail.) If we've defined a function `ϕYEARSRAIN*WettestMonthNumber` then we could write:

```
WettestMonthCount := LotsOfData*CountOccurrences( *WettestMonthNumber );
```

Which should give us an array in the same way that `GROUP BY` coupled with aggregate functions in SQL allows us to classify statistically. Here we'd be seeing where most wet months come. (We've skipped defining the data structure of `WettestMonthCount` and cheekily assumed that `*CountOccurrences` will be clever enough to produce something suitable. This also raises the issue of where built-in data structures such as arrays should stop and sets should begin.)

Here's a conundrum: Suppose we have a function that operates on single items in such a way as to change the state of the objects in a way that affects the overall set operation. For example where we're processing the set in a certain order and the operation fiddles with the order. Imagine processing items from cheapest to most expensive while increasing prices by 10%. In one possible implementation the first item gets its price inflated then *immediately repositioned* further down the list to happen again ... and again! It's an interesting question whether such operations can be proved to be safe by looking at the axiom dependencies.

There are traditional set-to-set operations - Union, intersection and not. These can be applied to data or meta-data. However to my mind these are equivalent to formal logic - *that we don't use in everyday life* to get real work done. It's handy to be able to ask

"How many black pawns have been captured" rather than struggle with "(intersection of black and pawn) intersection (not on\_board) ... now count them". We're much more likely to want to discover and structure facts by cross-reference - that's how relational databases work and they're quite popular! Suppose I have a number of salesmen and a number of product types - What I want is to analyse their performance perhaps by creating an 'intersection' of sales in the form of a table with person down the side and product type across the top. Clearly this isn't anything like an 'intersection' in the classical set sense but as it's a common requirement it ought to be part of our repertoire. Let's see what the code might look like:

```
// Defining sales records structure
ϕSALE.Salesman := ϕSALESMAN;
ϕSALE.Product := ϕPRODUCTTYPE;
ϕSALE.Value := Currency;
ϕSALE.Date := ϕDAY;
ϕSALE*InPeriod( Year, PeriodNo ){ ... (useful code) ... }; // true if date falls in specified period
// Now create a set of sales and add lots of data to it...
// ... so we can analyse it
Analysis := Sales*Intersection( .Salesman, .Product, *Total(.Value), *InPeriod(2007, 2) );
```

In the last line we're defining the axes of a two dimensional table, what to put in the cells and a selection condition. This looks a bit spreadsheetish which to my mind is encouraging. We've got many-to-1 relationships in our property values which seems just fine.

## Chamaeleon meta data

Now suppose our salesmen concentrate on a small section of our products and instead of a table we want a list of product types with sub-records for salesmen (ie in a tree layout) or a salesman-by-salesman list with a breakdown of the types each has sold as a sub-list, how might we achieve that result? Actually all we have to do is read across the rows of the tabulation in turn or read down the columns in turn. This is merely a matter of presentation.

Where we do have to re-think is if we've got a different meta-data structure. The tabulation we did was simple because we could cross tabulate the two key properties. But suppose ϕSALE was not so central and not so rich:

```
ϕSALE.Value := Currency;
ϕSALE.Date := ϕDAY;
ϕSALE*InPeriod( Year, PeriodNo ){ .....}; // true if date falls in specified period
```

And suppose our sales were heavily compartmentalised by product type within each salesman within each sales region. ie A sale 'belongs' to a product type which belongs to a salesman who belongs to a region. This looks like some nesting of sets. Note the extensive use of the colon to indicate relations:

```
// set of sales regions
Regions := $SALESREGIONS;
// having a number of salesmen
ϕϕRegions:Salesmen := ϕSALESMAN;
// each salesman has a number of accounts - 1 per
product type
ϕSALESMAN:Accounts := ϕACCOUNT;
ϕACCOUNT.ProductType := ϕPRODTYPE;
// each account will accrue sales
ϕACCOUNT:Sales := ϕSALE;
```

It is no coincidence that this nest of relations puts us in mind of a set of tables in a relational database.

Now when we make a sale we put it in the right account of the right salesman of the right area. How can we analyse sales now? At the lowest level we could total the sales for a single account

```
AccountTotalSales := SomeAccount:Sales*Total( .Value, *InPeriod( 2007, 2 ) );
```

which in English says : Look at the set of sales for some specified account and apply a totalling function over all the items in it - if in the appropriate period. The important thing to note is that :Sales is a Set... ..which will have a ghost class ... which we might be able to use in order to nail down this vague 'SomeAccount'.

```
ϕϕACCOUNT:Sales*ValueInPeriod( Year, Period ):= {
    result := *Total( .Value, *InPeriod( Year, Period ) );
};
```

That's given us a way to deal with the sales within an account, now what about all accounts for a salesman?

```
Analysis := SomeSalesman:Accounts*Tabulate(.ProductType,, *ValueInPeriod( 2007, 2 ), True );
```

In English : Over the set of accounts for some salesman perform a one dimensional tabulation function. Categories will be the product types encountered while the data will be the sum of the ValueInPeriod function for each account for February 2007. With any luck this should produce a two column table with products down the left and sales on the right. eg

Bathroom fittings	£3456.78
Central heating	£7765.66
Boilers	£2995.00

(This time we've incorporated the eligibility test inside the totalling function.) We can think of this as a 'baby' version of the intersection above.

However we're not home yet and need to repeat this by tabulating this result at a higher level. What a bore and how long will we have to spend debugging our code?

Fortunately there may be a way out that uses the fact that *if A is related to B then conversely B is related to A*. Let's try this in code:-

```
FebSales := $SALE*InPeriod( 2007 ,2 );
Analysis := FebSales*Intersection(
    <($ACCOUNT.Sales).ProductType ,
    <($ACCOUNT.Sales)<($SALESMAN.Accounts).Name ,
    *Total( .Value )
);
```

The first line says "FebSales" is the set of all objects with the class ϕSALE that match the criterion of returning true to the InPeriod function.

The second part introduces reverse-lookup (using the syntax '*<(...)*') to get the first two arguments to be used as labels for rows and columns. In English: [line 2] Perform a tabular intersection over the nodes in FebSales. [Line 3] Look at the objects in \$ACCOUNT (ie the set of all objects with class ϕACCOUNT) for where this SALE node appears in the :Sales relation, now with that ACCOUNT node return the .ProductType property. [Line 4] Now repeat that but daisy chain another reverse look-up in \$SALESMAN (set of all objects with class ϕSALESMAN) by seeking occurrences of this ACCOUNT object in the :Accounts relation and then return the name of the salesman. [Line 5] fill the cells of the table with the total values."

Now it may not seem a spectacular idea to relate in both directions but it is certainly quite neat to be able to write two statements to hack our way through a jungle of confused data relationships. Readers familiar with relational databases will see

similarities. Sets can be likened to tables with classes being their table definitions and '<' doing joins.

How should we implement 'reverse lookup'? Is this a feature that is automatically built-in to every relation that comes with all that maintenance overhead? I think we have to bite this bullet. With proper design it shouldn't be too much hassle, and with Gbytes of memory easily available it should be 'invisible'. However from a real-life implementation point of view I'm a little concerned about the scope for all these relations getting in a knot and not having the means of untangling or verifying the 'true' relations.

## Package, preparation and poison

There is nothing very new about having packages of classes and supporting resources, however there are issues we need to address in a system that is designed with the goal of enthusiastically devouring new meta-data.

Let us consider a business situation where new and updated classes are being made available to us. Let us suppose we have a working system with classes such as cPATIENT, cEVENT, cPRACTITIONER and so on, with easily dozens of classes. (We'll try to ignore interface and data-conversion issues and concentrate on the meta-data.) What if we're offered or encouraged to 'upgrade' to a new and improved cPATIENT class?

- Do we assess 'risk of a shambles' simply on the 'trustworthiness' of the supplier?
- What could go wrong?
- Is there any way of testing?
- Is there any way of backing-out if it goes wrong?
- Are there any differences if we're making home-grown changes?

At this point any experienced IT person will run away with the screaming ab dabs - especially since we're talking about a tangle of interrelated and slippery meta-data. It's bad enough when somebody suggests a change to a database schema but fooling around with this evolved spaghetti is too scary to contemplate.

It is tempting to say that humans don't seem to have much problem with adapting to altered concepts; but we do. For example my neighbour sealed up their letterbox which completely baffled me - it was days before I discovered they'd put a box for mail on the wall, in full view! Last night I left a message on my brother's answering machine with the phrase "nineteen-thirty-two" which baffled him until I later explained it was a time not a year. 'Brains run in our family' but once we've got the wrong end of the stick we hold on to it tenaciously. It should be apparent that something like changing the way a year is represented in a data field from two to four digits could have a similar baffling effect - but without the confusion surfacing until damage was done. Once humans know they're confused they can seek enlightenment. Also humans can spot what's missing, contradictory or just 'a bit peculiar'. It might be helpful to classify this as:

contradictory . . . .	meta-data: OK
	real-data: Bad
confused . . . . .	meta-data: Bad
	real-data: Untrustworthy

Can we say that purely supplementary meta-data, whether as whole classes or new properties etc. are 'harmless' from the point of view of not affecting any existing real or meta data?

- Tactically : Yes
- Strategically : No
- Realistically : No

Let's pick the bones out of this. Suppose you are going on holiday to Lusitania for the first time and want to learn a bit of the language. You pick up a battered copy of *Fox's universal phrase book for the adventurous traveller*. Is it harmless? Even if it doesn't cover any of the interesting phrases you want you haven't unlearned any English. So from that point of view it is just a harmless waste of time. Now your travelling companion has found a more accurate book *Be lucky in Lusitania*. Unfortunately you're going to have to unlearn pretend Lusitanian or get in a horrible mess. If on the other hand you get stuck with Fox's farrago of foreign phrases then you could end up at the police station trying to explain yourself - which using my book will only make it worse! One approach is to try a little at a time and see what reaction you get to build up confidence from experience.

The moral of this story is:

- you have to rely on trust when pushing out into the unknown
- try to experiment and be alert for problems

Which applies even to pure additions, let alone modifications. (Remember that some meta data might get modified behind the scenes as real data is being processed. It's a scary thought that needs looking at.)

Returning to the medical scenario: How many patients are we going to kill before realising there's something wrong? If past IT failures are anything to go by - lots. Clearly that's not good enough and we need some quality assurance system for meta-data.

## Definition of scope - a package

The first essential is that we know the extent and type of the changes being proposed.

How can an outside source of a class know what this is? Possibly if they supplied an earlier version of the same thing, but even then there's no guarantee that our system is still in exactly that theoretical state. The inevitable conclusion is that we need a tool for being able to discover differences between two classes (actually much more than classes), or describe what a class (or other elements) actually do. This means building and comparing maps of the meta-data. ie meta-meta-data.

We are going to need a proper listing of what is being supplied. For security we ought to base this on the actual classes being supplied rather than a separately concocted manifest.

## Change management

Let's suppose `CPATIENT` evolves along these lines

- MkI starts with a property `.BloodGroup` which is of type `String`.
- Then MkII is offered with `.BloodGroup` as member of the set of labels `{£A,£B,£AB,£O}`;

- Then MkIII arrives with `.BloodGroup` as `ϕBLOODGROUP`.
- Now MkIV forces `.BloodGroup` to belong to a closed set `$BLOODGROUP`
- Finally MkV upgrades `ϕBLOODGROUP` to add Rhesus factor

We might tabulate the single-step changes

I to II    Type of `ϕPATIENT.BloodGroup` changed to restricted-label  
 II to III    Type of `ϕPATIENT.BloodGroup` changed from string to `ϕBLOODGROUP`  
 III to IV    Type of `ϕPATIENT.BloodGroup` changed to restricted-class  
 IV to V    `ϕBLOODGROUP.Rhesus := {'Pos','Neg'}` added

### Data compatibility

By the time MkII comes along we already have a number of patient records. How do we check them for compatibility or convertability? When we were code walking we were in the meta-data realm now of course it's real data that we're being asked to change. Can we 'rewrite history'? Ouch! There might be all sorts of limitations on the convertability of data and conversion rules might change depending on the application or the personal preference of the system operators.

Suppose we have a data file of MkI `ϕPATIENT` records. Where `.BloodGroup` was "A" (a go-as-you-like string) we can easily convert that to `fA` (Label-A) from the enumerated type but what happens if the field value is "A+" or "n/a". We simply don't have an acceptable choice from the set of labels we've been given and also we're not in an interactive validation context, say an input screen, where this matter can be dealt with by someone who knows all the circumstances. In this example the restriction prevents us creating a proper patient MKII record. Normally we'd expect the programmer to allow a 'not-known' value, but even that doesn't answer what we do with "A+". What's really happening is that we've got conflict between real-world and meta-model as we've come across before. So perhaps we have to hack the `ϕPATIENT` class to allow the `.BloodGroup` property to be NULL or perhaps we have to hack the data or perhaps we have to go back to the drawing board and think again what we're really trying to achieve.

This discussion throws up two key concepts:

- We need conversion methods. These must accurately identify which version of a class is being converted to another.
- A way to generically flag 'Defective object' where conversions fail.

There will occasionally be times when we are quite happy to downgrade an object from one class to another. Perhaps in the process of storing or transmitting objects we remove some live information which may or may not be reconstituted later.

It should be clear that we need to be able to do this on a trial basis with some sort of exception report to indicate feasibility. (It's getting quite serious, but this is what practical IT systems should be all about - if they're not being skilfully and diligently managed then there's a problem.) We asked above what difference there would be if the changes were home-grown? Presumably the local changes are made for a reason, that is the reason and understanding comes first rather than as an accepted fact when the external package of changes arrives. All of this points to:-

- Having 'cells' of compartmentalisation which are simple enough to be understood with a small amount of study of the appropriate documents. This is equivalent to

having many drawings and sub-assemblies for logistical, functional and design purposes in any manufacturing environment. Cells might be nested.

- Having people who have the job of looking after the mechanism who understand the technology and can use the necessary maintenance and diagnostic tools.

This is in contrast to the modern day pressure to wrap everything in a black box and forget about it if possible. Perhaps there is good news here: Perhaps it is a really good thing for a business that depends on information technology to have people that know how to make the most out of it - not only to keep the 'machinery' running efficiently and reliably, but also to promote useful developments.

The extent to which version conversion can be automated is an interesting one. There are certain simple rules as used for casting types in classical computing languages, but what about where a new property or relation is introduced. Suppose we introduce a relation of :Relatives to our  $\phi$ PATIENT. There will no doubt be such relationships in our database but probably no way of finding them automatically. The danger here is that when we make an enquiry and find a blank we'll assume that there simply aren't any as 'no information' becomes 'none'. It's a horrible feeling having patchy and unreliable data. In this case we might want to mark the property value 'Defective' (and possibly the object itself) until patched up in some way or other. Well, at least we know what we don't know. This is a big subject which needs researching in detail so for the time being I'll simply propose:

- Data objects and components to have a built in 'defective' flag that can be raised when there is a conflict with the meta-data.
- Classes that replace or upgrade others have a formally specified procedure for dealing with data anomalies. These procedures to be humanly reviewable and switchable.
- Meta-data discrepancies to be discoverable by automatic means (and compared with the upgrade procedures.)
- Absolute ID for classes is not essential given the last mentioned item, but could be desirable for clarification of documentation and tracking modifications.
- There should be standard methods for automatically converting between classes provided by package authors. These may need to 'be approved or tweaked' before use on a particular system. (This has to be bi-directional - see below.)

Also there is a need, possibly a very general one given the shifting-sands nature of the meta-data to be able to undo changes. One way might be to snapshot real data and meta-data.

Two more interesting sidelines to the issue of adapting data to suit different data models are exposed if we consider two systems running different versions of meta-data that pass real data between them.

- It may not be possible to do a 'big-bang' conversion for all data on both systems.
- What is the definition of 'upgrade'? Is there a conversion procedure for converting say a MkIV  $\phi$ PATIENT to a MkII when the more developed system sends data to the less advanced one?

As the answers to these questions need a lot of debate and research we'll leave them as matters that need clearing up. In the meantime we might want to qualify a class by the realms ("cells" was the term I used above) in which it is 'certified' to operate. This will enable 'internal', 'interface/shared' and 'wild' classes to be identified.

## Code walking

What should we do if (somewhere in our acres of code) we have `SomePatient.BloodGroup := "A"`; which is valid only for MkI and MkII? The first thing is to find the hotspot

- Where does `.BloodGroup` get used
- Check use against actual or proposed class

Is this possible? Yes I think so. In a traditional compiler statements are checked against what is possible, here we're slightly *reversing the checking* by asking is the proposed class compatible with our code - and data!

## Fixing the code

Hmmm... For sake of argument let's assume that we have a magic data browser that is clever enough to automatically adjust the user interface and data structures. That leaves custom code that could be broken, or it could use classes in a particular way that makes sense to us but the authors of the upgrade didn't think of. In our `.BloodGroup` context we could have been happily using MkI `ϕPATIENT` which allows any string as a property value for `.BloodGroup` to show both group (A,B,AB,O) and Rhesus factor (+,-) and Don't know/don't care and combinations. So if we have live and useful code

```
SomePatient.BloodGroup := "A+";  
OtherPatient.BloodGroup := "Due from lab 12/10";
```

we simply don't want to upgrade and lose our useful functionality. We need a MkVI which allows a comment before we get out of the one-step-forwards-two-steps-back upgrade trap. (In this specific example of blood types using "+" and "-" is known bad practice, so there may be 'good reasons' for wanting to upgrade from MkI - But that doesn't remove the problem caused by upgrading. )

## Can we allow automatic (or any) polymorphism?

In this particular case we *could* get round a lot of our problems by letting `.BloodGroup` be either a String *or* whatever the later marks wanted it to be. Is this a good idea? Suppose we look at this from the point of view of the data in the property value trying to be processed in a 'MkV world'. "Hey I'm not compatible with all this new-fangled stuff, why can't I be processed as in the good old days?" There's nothing stopping us building-in a fall-back mechanism for problem cases that works back through layers of upgrades until it finds an old version of the class it could use ... But I think that's horribly complicated and after time practically random in operation being almost untraceable through layer upon layer of overlapping changes.<sup>7</sup>

What about if the upgrade for a particular data item fails and is marked with the Defective flag? Presumably we don't upgrade whatever was defective because we can't. Whether this can be lived with is a matter that will need consideration by the system operators. There is also the question of where the 'this component is defective so the whole thing is defective' process stops. If `ϕPATIENT.BloodGroup` is the only sticking point then do we have to make all offending instances of `ϕPATIENT` as still 'MkI (Defective)' or just the property? We asked this question above and decided it needed an answer but more research.

---

<sup>7</sup> But what an interesting research project it would make to explore how a system behaves when it 'gets confused' and has to seek out alternative methods.

My gut feeling is that polymorphism should not be allowed as such. We can get roughly equivalent flexibility but with formality from Ghost classes derived from Sets containing a miscellany. (By the way, this also means that if we know where a Ghost class is used - which the system should be able to discover, then we can check the validity of adding a strange class into it to see if it breaks any code.)

## **Package**

Interactive tinkering with the meta-data from a console is one way of developing the sophistication of the system and checking for inconsistencies. This is a very convenient method for discussing and exploring language features but not so hot when there's a large body of meta-data or real-data to be installed on a production system. As we've seen there are all sorts of conversion and compatibility issues associated with new or improved meta-data or real-data and we need a way to manage these changes.

The obvious way is to compartmentalise the changes so they can be identified, documented and tested and approved locally. This is bound to be more than clicking on an 'install now' button for reasons discussed above - we need to understand the implications of the changes. A package might not be completely self contained - relying on ubiquitous classes and established meta-data.

The concept of a well documented package of code with dependencies is not new, however we might want to add a couple of twists.

- Checking dependencies and conflicts remotely before installation. (Using the comparison/analysis tool mentioned above.)
- Maintaining a boundary between this meta-data and real-data after installation so that the package runs as a sub-system with declared interfaces or publicly accessible classes and data. From the outside only 'connector classes' might be visible giving a particular slant on the data and meta-data.

The architectural implications of this last item are quite significant.

## **Poison**

Finally we need to be alert for poison packages or interactive injections. It would be quite easy to disrupt meta-data and quite hard to trace what had happened. There is no reason for this to be malicious but the effects could be serious.

If there is no antidote then we need to be able to wind the clock back to before the disruptive changes were made. Either this involves snapshots or a change log or both which will have to be implemented in the system as a matter of course.

# The representation of knowledge

## Part 3 - Conclusion

We have seen that the basic concept of combining classes with sets seems to hold together quite nicely. There appears to be a pleasant synergy between explicit class definitions and dynamic evolution of datasets. There are interesting analogies with relational databases.

### Basic principles - classes

1. A class in the object oriented sense is a backbone for properties, relations and methods. Relations are multiple pointers to other data objects.
2. Classes are (normally) mutable and able to deduce a lot about themselves by inference.

The following is all that is required to tell the system there's a class( $\phi$ ) called PERSON with a constructor function(\*) and a name property (.)

```
 $\phi$ PERSON*New(Name)={.name=Name;}
```

At this point there is no type restriction except that the argument Name must be castable to the property .name.<sup>8</sup> If we now create an instance using a string the class will 'learn' these are strings.

```
john =  $\phi$ PERSON*New("John Brown");
```

We can explicitly alter the person class on-the-fly:

```
 $\phi$ PERSON:parent= $\phi$ PERSON;
```

Which says "There's a relation(:) called parent which will link to zero, one or many PERSON data instances."

Now we might want give the *data instance* john an age property which will

- either enrich the PERSON class
- or create a new derived, possibly anonymous, class.

```
john.age(42);
```

```
 $\phi$ AGEDPERSON(john);
```

```
john.age(42);
```

These are the slippery and clunky extremes of an adaptive knowledge system. The key point is that the way the data is used affects the structure of classes.

### Basic principles - sets

3. All the data instances of a class belong to a set with the same name. This is similar to how the records in a database are associated with a table definition.

---

<sup>8</sup> Case is not significant here. Classes are capitalised for ease of comprehension.

4. Any set has a collection of properties and methods derived from its members. These are called "Ghost classes". The capabilities of ghost classes are an intersection of the classes of all the members.
5. Operations can be applied to all instances in a set. However valid operations are determined by capabilities of the ghost class.

The following explicitly creates two sets(\$\$) and one class. These in turn create implicit matching ghost classes(φφ) and implicit \$PERSON set.

```
john = φPERSON*New("John Brown");
mary = φPERSON*New("Mary Brown");
jane = φPERSON*New("Jane Brown");
alan = φPERSON*New("Alan Brown");
$parents = {john,mary}
$children = {jane,alan}
```

Now we can develop the classes in different directions

```
φφparents:children=φPERSON;
```

which says "The ghost class 'behind' the set \$parents now has a relation(:) called children which points to zero, one or many PERSON data instances.

```
φφchildren.favouriteToy=String;
```

which adds a property specifically to the members of the \$children set.

There are various ways of connecting the parents to the children

```
jane:parent += john;
jane:parent += mary;
alan:parent += john;
alan:parent += mary;
```

Or more succinctly

```
jane:parent = {john,mary};
alan:parent = {john,mary};
```

Or more succinctly

```
jane:parent = $parents;
alan:parent = $parents;
```

Or more succinctly

```
$children:parent += $parents; // operates over all members of the $children set
```

Interestingly we do not need to have previously defined the φPERSON:parent relation as we did above (φPERSON:parent=φPERSON;). Doing so is probably a good idea, but if we don't the system will be able to determine what (so far as history teaches it) the :parent relation points to. (The reverse relation will be shown later.)

At this point we can count the number of instances of φPERSON, that is the members of \$PERSON but that's about it. To illustrate the nature of ghost classes we'll use another example.

```
// define three classes and constructors for them
φSHAPE*New(Name,Area,Sides)= { .name=Name;
                              .area=Area;
                              .sides=Sides;}

φRECTANGLE=SubClass(φSHAPE);
φRECTANGLE*New(H,W)= { .name="Rectangle";
                      .height=H;.width=W;
                      .area=H*W;.sides=4;}

φCOUNTRY*New(Name,Area,Population)= { .name=Name;
                                       .area=Area;
```

```
.population=Population;}
```

```
ϕCOUNTRY*Density = .population / .area;
```

At this point note that ϕCOUNTRY and ϕSHAPE are completely unrelated classes.

```
// create some data (Using metres/sq metres)
plate=ϕSHAPE*New("Dinner plate",0.03,1);
plot=ϕRECTANGLE*New(15,23.6);
portugal=ϕCOUNTRY*New("Portugal",92345000000,10848690);
// and stick into a set
$bitsNbobs={plate,plot,portugal};
```

Although the bitsNbobs set has data items belonging to unrelated classes in it we can still do operations on the whole set. The ghost class ϕϕbitsNbobs consists of the intersection of all the component classes. In this case it will know about .name and .area. So how about

```
tiny = Smallest($bitsNbobs.area);
table = Tabulate($bitsNbobs.name,$bitsNbobs.area);
```

Now we can work explicitly with the ghost class to define a new function:

```
// instead of sq metres return square miles
ϕϕbitsNbobs*SqMiles=.area / .000000385;
```

Now you're all asking about the 'overlap' between ϕCOUNTRY and ϕSHAPE. Firstly there may well be good reasons when we want to work with what we've got rather than force everything to have to derive its properties and methods from a single ancestor. Secondly we have lots of other possibilities for structuring with significant differences.

- Sub-class

```
ϕCOUNTRY=SubClass(ϕSHAPE);
ϕCOUNTRY.population=Number;
```
- Wrap

```
ϕCOUNTRY.shape=ϕSHAPE;
ϕCOUNTRY.population=Number;
```
- Relate

```
ϕCOUNTRY:shape=ϕSHAPE;
ϕCOUNTRY.population=Number;
```

This is more useful where there are multiple things to relate to. For example Olympic medal winners.

## Language features

6. Sets can be enumerated and indexed.  
The necessary hooks to use a set as a vector should be built-in to provide single item or item-by-item access.
7. Labels (ℱ) are used to reliably abstract identity and may be thought of as enumerated types.
8. Relations (:) automatically work in reverse. A 'what references this' feature.

Suppose we want to enumerate days of the week. We could be primitive and do this with an array of strings. More advanced would be an array of ϕDAYOFWEEK data instances. More advanced still would be converting the array into a dedicated ϕDAYSOFWEEK object with an index and the appropriate item retrieved by number or string:

```
tomorrow = Find($DAYSOFWEEK, .name, "Tuesday");
```

or

```
yesterday = Index($DAYSOFWEEK, today - 1);
```

These may be appropriate for some collections but really days of the week are firstly fixed and secondly have an abstract identity. So we provide a way to define fixed-index-label sets as illustrated below using f.

```
// specify the class attributes
ϕDAYOFWEEK.ShortName = String;
ϕDAYOFWEEK.Abbreviation = String;
ϕDAYOFWEEK.IsAWeekend = Boolean;
ϕDAYOFWEEK*New( S, A, W ) = { .ShortName=S;
                              .Abbreviation=A;
                              .IsAWeekend = W; };

// now fill set of DAYOFWEEK with labelled nodes
$DAYOFWEEK£Monday = New("Mon","Mo",False);
$DAYOFWEEK£Tuesday = New("Tue","Tu",False);
// etc for all seven days

// and now we can use this as
tomorrow = £Tuesday;
```

This has the advantage of separating the label identity of something from the data value of the contents. It also means we can 'compile' these as constants. In the example we've left out the "\$DAYOFWEEK" before £Tuesday assuming that labels are unique and thus unambiguous, however this may be a convenient shortcut rather than reliable practice. This is pretty much the same thing as class methods in OOP. We can also imagine class methods for finding say the 'day before'.)

With the example of the Brown family above we populated the ϕPERSON:parent = ϕPERSON; relation by giving the children pointers to john and mary. A reverse relation asks "what points here?" or "where does this object appear in another's relation?" We could code this as:

```
johnsChildren = john<($PERSON:parent);
```

A complex web of double-ended pointers will evolve, but that's the price we need to pay to be able to extract relationships which is what knowledge is all about. With Gb of cheap memory this shouldn't be an issue.

## System functions

We'll need a console, persistent storage, ways to transfer data between persistent storage and volatile memory, ways to interface with other systems. Beyond this we need an inspector and auditing mechanism.

## Issues

Allowing data to influence data structures is quite a slippery business. Inappropriate or unexpected data values could improve or break logic. This is more frightening when considering 'a standard upgrade' for a number of live systems which may all have evolved differently.

To manage complexity we'll have to break systems into manageable chunks. This impacts upgrades and alterations as mentioned in the previous paragraph. This implies an identity system such as unique namespaces, specified dependencies and interfaces. Bearing in mind that the internal operations of a module might be adapting as time goes by but the interface will 'stay the same until changes are explicitly propagated'

we'll need to give some deep thought to the way in which outside processes are connected to private internal ones.

## **Postscript**

This paper was originally prompted by my requirement to store simple application configuration data - followed by a realisation that a few trees of data and the odd data definition was looking at a solution without properly understanding the problem. When a bit of experimenting showed that the object oriented approach and the relational database approach could be easily merged it looked like there was something worth investigating further.

Building a system that can work things out for itself and which is frighteningly susceptible to picking up knowledge based on a small set of easily implemented rules is very appealing. I suggest the next job is for somebody to knock together a prototype and see what the practical issues are...

...and see if it can actually be used for something useful.