WORKING NOTE AC129

Alan Creak
2001 January 8

PROPERTIES OF CODE AND DATA

I point out some parallels between descriptions of code – particularly source programmes – and of data – particularly disc files.

PREFACE.

I started writing this note on the day I was struck by the brightish idea reported herein. Any mention of "today" is therefore to be interpreted as 2000 September 9.

As is not uncommon with brightish ideas, once you get past the first bit, things get harder. In trying to record the insight comparatively formally and comprehensibly, I found no dearth of material to work on, but encountered a problem in taxonomy : how should it all be classified ? Now I don't want to spend any longer on this note, but haven't got as far as I'd hoped. Well, I suppose that's what the notes are for, really.

So the current state is : I'm still impressed by the idea, but I think there's a lot more to do to work out the details. Do not watch this space, but further developments are not impossible.

THE IDEA.

Programmes and data are both kept in files. In both cases, the raw material – numerical or symbolic data, or machine code – is associated with a quantity of metadata, connected with the raw material itself or with its administration in the computer system.

It is suggested that *in both cases the metadata can usefully be organised into essentially the same structure*.

This will hereinafter be identified as the Idea.

BRIEF HISTORY.

Sometimes I'm a bit slow. It has taken me well over ten years to notice the Idea after becoming cognisant of the things connected. My lapse becomes all the less explicable when seen in the light of my long-standing, and oft-repeated, emphasis on the dual relationship between code structures and data structures. If I have a defence, it's that I haven't heard about the Idea anywhere else, though to be fair I haven't been looking in places where such things might be found. (Are there such places ? Where ?)

I began to muse over the file side of the question perhaps around 1986, when Macintosh systems began to infest the Computer Science department, and with them the strange files[1] with data and resource "forks". I mused because I like things to be orderly, and perhaps therefore wanted to see more in this development than just another way of storing stuff. This led me (perhaps around 1990, but I don't have the old records any more so can't check) to expand my discussion of files in the Operating Systems course to allow the possibility of several components.

Perhaps I had been alerted to the possibility of such extensions by some earlier experience of stuffing different sorts of information into files. In 1979 I was involved in the development of Zeno[2], which was intended to run the computing services for students at Auckland University. Having in mind that services for students would certainly be offered on several different machines, we hoped to provide a consistent interface which would work on all of them. Zeno was our answer. We thought of it as an operating system "mask", which would use the facilities of the machine's native operating system to provide a comprehensible set of services, which themselves would be designed to be better adapted to a learning environment than were the standard compilers and other software. This led us to require that various files should be simultaneously "native" files, conforming to the host operating system's conventions, and "our" files, with additional information appropriate to our requirement. A first design[3] had all "our" information contained within the files, merged in and stripped out as was necessary, but further analysis[4] led us to propose that "our" information should be held in a separate, but associated "profile". The profile is supplementary information about a file which is neither an attribute of the file as a whole nor the file data proper, but was nevertheless clearly a component of the file in some sense.

I've presented the preceding paragraph as a file topic because that's how it seemed to us at the time; alert readers will have observed that it could be just as readily seen as a programme topic. In hindsight, that reinforces the Idea – but we are still concerned with history.

The programme side of the question emerged earlier still. I learnt and used Fortran in 1962, or so; I bought a Cobol manual around 1967 and first used it in anger around 1970. I knew about machine language even earlier – perhaps from 1955 ? – and used assembly language seriously in 1969. (That doesn't include a spell of programming an Olivetti Programma 101 – perhaps 1966 ? – which was the best way I know of learning the principles of assembly language, though I was never able to convince anyone else of that.) Around 1970, I knew Fortran and Cobol reasonably well, knew a bit of PL/I, and had read about Algol. I accepted the Algol-Cobol-Fortran trio as different and comprehensible answers to rather different varieties of the same question, and found PL/I rather clumsy.

It was perhaps around then that I became aware that Cobol was looked down upon by "academics". (I use the quotation marks to suggest that the title is not deserved. I was, and am, an academic, and could not see the force of their assertions. Today I have become even more sure that they were wrong.) Cobol was the only one of the trio to provide facilities for managing characters in straightforward ways; more significant for present purposes, it was also the only one to take account – in the Environment Division – of the possibility that it might be used on different sorts of machine. Both of these provisions seemed to me to put Cobol at a higher level than the others from the point of view of sensible design, but the provision of the Environment Division[5] goes beyond immediate programming convenience to provide a new sort of information. This is associated with the executable code, but complementary to it, and essentially independent of it; in principle, and fairly reliably in practice, to move a programme from one computer to another it is necessary to change only the Environment Division.

I don't know when I really understood that declarations and executable instructions in programmes were different in nature. I suspect that I was not alone in this; I recall earnest debates on whether declarations should be permitted within the body of programmes, and what their significance would be if they were. I knew that they had different functions, but thought of them all as fitting into the single pattern of what had to be done to make the programme go. Declarations then became mainly instructions which were executed to provide storage space (so it was not silly to think of declarations in the middle of procedures), and they became administrative operations, rather similar in nature to procedure calls. This view is probably fostered by an interest in compiling methods (which I had), though challenged by an interest in programming theory (which I hadn't).

The question came to the fore around 1975, when, with colleagues in the Computer Centre, I was engaged in building a Basic interpreter for a minicomputer[6, 7]. As there was no recognised standard, we began by inspecting all the Basic manuals we could find, and it was wondrous to behold their variety. We didn't really like any of them, and one of our reasons was the apparently universal practice of deciding on the language semantics by picking a convenient implementation technique. We thought we should decide on the semantics we wanted, then implement a system accordingly. Details appear to be lost in the mists of time, but the final decision (there were reasons, but they are not relevant to this discussion) was to interpret the programme strictly according to the execution order as determined by line numbers and transfers of control as directed by various instructions. Declarations were not regarded as special, so their significance depended on where they were executed in the programme.

That gives the wrong answer – that declarations are just ordinary instructions – for this discussion, but the Basic experience led me to think a lot more about the significance of declarations, and I began to see them as indeed declarative items within a procedural context, different in nature from the executable instructions, while obviously closely connected. This separation becomes much clearer in later languages (Algol 68 ? – someone stole my book; Ada ? – I've forgotten) where notions of information hiding became common. Once again we have collections of data (declarations and procedural code) which are different in nature, but intimately connected. In this case, the conventional programming language syntax imposes a relative order on the two parts which has almost no significance, and in some ways obscures the association of the declarations with their possibly broad scopes.

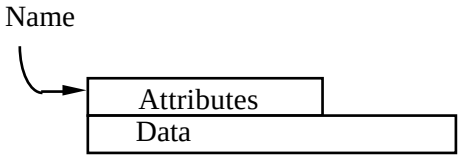
But it was not until today that I noticed the parallels between these two trains of thought.

ABOUT THE FILES.

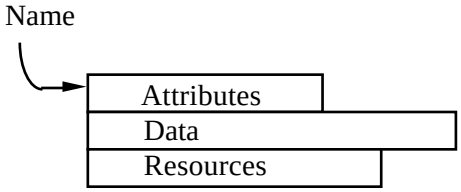
I deal with the files first, because that's where most of the agonising has been, stimulated by the demands of the Operating Systems course. The first step was to make sense of the Macintosh files.

A Macintosh file[1] has three components : attributes, data, and resources. The resources identify things which will be required when the file is used, but which are usually considered to be in the nature of comparatively static appendages – as opposed to the data, which may be read and written as usual. The resources are typically independent of the file in many senses, so they can be changed without affecting the significance of the file itself. Examples are icons, details of presentation of various sorts of window, sounds used, and so on. This additional dimension to the file system requires appropriate supporting software, so special procedures to handle the "resource fork" are supplied in the Macintosh system, and there is a "resource editor" to manipulate resources in various ways. Here's a diagram comparing the Macintosh and conventional file structures :

CONVENTIONAL



MACINTOSH

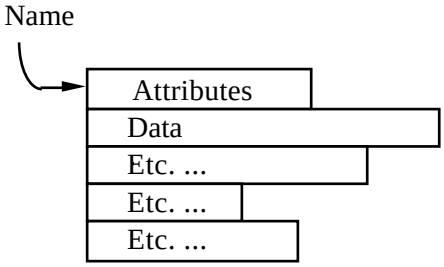


Does that exhaust the possibilities ? Certainly not, once you start to think of it. The Zeno profiles[4] fit the pattern exactly, but in this case the additional material runs parallel to the file data, and is specifically connected to it at various points. For a more conventional example, security properties could well be seen as sufficiently specialised and sensitive to be kept somehow separate from other attributes; an access control list[8] is already handled in a comparable way, without the built-in connection to the associated file.

Formatting information for a data file can be seen in the same light. The information required to define the format of, say, a text file is quite specific. In a text-formatting programme, it is commonly expressed as a collection of styles which can be invoked and associated with various parts of text as required. In this case, the text and the styles are quite distinct and independent components, and the links between them form a third component. In practice, all these data are commonly kept in the same file data component, with somewhat different separation methods for mark-up notations (where the styles themselves might be in a separate file) and "word-processors". In both cases, it can be less than easy to extract the basic text from the file, particularly if you do not have the precise software concerned.

It is not unreasonable to generalise the "Macintosh" pattern, and to regard a file as a collection of an arbitrary number of separate, though related, sets of information, which I shall call (indeed, already have called) *components*. A structure of this sort would be not too difficult to implement in a rather general way, which would leave room for expansions of many sorts. (It begins to sound distinctly like an object, as in object-oriented systems – particularly as resources can include executable code. You can think of that either as an amusing accident, or as an indication that the object model does indeed capture something fairly fundamental about the nature of computing.) In terms of the diagram above, the general structure would look something like this :

GENERALLY ?



Just what appears in the "Etc." areas depends on circumstances. Traditional file systems provide nothing at all; Macintosh systems provide resources. Generally, there's no very obvious reason why explicit system support for specific sorts of component should always be necessary, nor that it should always be the same for all files in a system.

We preached this doctrine for many years in our operating systems course, where it grew naturally from our top-down design[9]. The world, as usual, is a bit slow in catching up, but it's getting there. Something almost identical with this structure has been reported[10] as a means of coping with files for high-speed parallel file systems. In these files, the data are organised into several separate components (which the authors unfortunately call forks), and different forks an be used for different purposes.

ABOUT THE PROGRAMMES.

I do not really know how it was that I came to compare the structures of programmes to the examples above. It happened while I was contemplating those examples, and expanding them along the lines I describe below. I had got as far as files with no attributes, and was pondering the IBM1130 Disk Monitor System's files which had just two attributes, and I was trying to remember what they were. I thought one of them was executability – and the rest, somehow, followed.

A programme usually has a name. It always has some sort of code, or nothing will happen. Unless it is written in a very low-level language, it has information about the code, including at least the information which we usually call declarations. Though they are commonly written as part of the programme and included in the same syntax specifications, this is accidental, as they are not (usually) executed in the same sense. If we had kept them separate from the start, we might have been spared a lot of silly argument about whether declarations can be scattered through the code, what it means if they are, whether you can put executable parts (such as computations of initial values) in the declarations, and so on. The questions wouldn't have gone away, but they would have been separated from irrelevant considerations.

Is there anything else ? Yes : the Cobol Environment Division[5]. This is not about what the programme does, nor about how the programme does it; it is about the programme's links with the world outside. Other examples can be found in the special features provided by different implementations of languages – such as the "project" structure found in the Borland implementations of languages in the C and Pascal families[11].

As a more general example, consider Knuth's "literate programming" software, Web[12]. A Web file includes both programme code and commentary, and there is software (Tangle and Weave) which separates one from the other for compiling or printing as required. The same pattern of two separate but complementary streams appears once again.

And, just to link things together, recall my earlier observation that our Zeno profiles, though we saw them as useful with many sorts of file, originally turned up in the context of programmes. A particular example was the provision of assistance with programme diagnosis and correction; we had intended to provide means for merging compilers' error reports with source programmes so that the two could be edited together and a new source file produced comparatively easily. This is a commonplace interactive activity now, provided as a matter of course by programme development software, but in 1979 it wasn't.

It is mildly interesting that Zeno offers another example of the sort of material that could well be kept in a separate structure – indeed, it is just the sort of thing that is managed well by the Macintosh resource forks. A principle of Zeno was that it would be menu-driven, so it was clearly a good idea to have a standard way to manage menus. We recognised that the menus should be handled as distinct entities, but thought that the details should appear in the context of the programme which uses the menu, so the complete separation of menu from programme found in the current Macintosh and Windows APIs was avoided. It is fair to say that the alternative was cumbersome; the menu specifications, written according to a special syntax[13], were embedded in the Zeno source code, and translated into further ordinary source code using a preprocessor. (For practical reasons, Zeno was written in Fortran.) The menu specifications, essentially declarative in nature, were converted into tables which were then passed to a standard menu subroutine for display and for managing the response[2]. Cumbersome or not, it worked, and it made the point that menus were not the same sort of thing as ordinary procedural code; the resource fork is an elegant solution to our problem.

THE CONNECTION.

Now, at last – what was it that I found out today ? It was that these two patterns were, in effect, one. Consider this table, which gets more speculative as it goes on (this is the taxonomy problem) :

	Files	*Programmes*
Name	Name	Name
Local attributes of name	Location Icon Size Date	Location Icon Size Date
Attributes related to using the name	Security	Security
Body	Data	Code
Internal attributes of body	Record structure Consistency criteria	Programme structure Declarations
Links between body and external entities.	Creating programme Formatting instructions	Source programme Cobol Environment Division Interface (Displays, Menus, API) Project structure
Material auxiliary to the body	Documentation Styles	Documentation Help
Material auxiliary to parts of the body	Annotations Zeno profiles Web text	Comments Error reports

Perhaps the name, the body, and their immediate attributes are more significant than the others. These are all things which have to be there (but see the next section), not incidentals which are optional (perhaps colour screen, sound) or can be changed without affecting the viability of the item (perhaps specific devices used, identity of input file) – though in some cases the items in each "perhaps" might be true requirements. I should remark, though, that I do not think that any of the items which I've included in the table are trivial.

This taxonomy of attributes is significant because (if I've got it right) the various parts are in some sense independent, and can be operated upon by their own specific programmes. The Macintosh resource editor is a simple example. So is the Zeno menu preprocessor, and it would have been a lot easier with a structural separation between the menus and the Fortran code !

WHAT DO WE DO WITH THEM ?

Clearly, as we are getting by without worrying significantly about this structure, we do not *need* to do anything at all about it. You could say the same about the wheel; the interesting question is whether we could get by better if we paid more attention to this, or a related, structure. Could we ? I don't know, but the fragments I've mentioned suggest that if we do recognise the existence of distinctions between different sorts of information pertaining to a file or programme then we can find better ways of doing things.

In current operating systems, we are usually provided with the basic file body and a fixed attribute list. Not surprisingly, the attribute list contains the information required by the operating system, which can sensibly be kept separate from everything else because they are used for different purposes. My point in this note is that we commonly wish to define what are, in effect, our own attributes, for precisely the same reason. Of course, we can get everything in the file body if we want to, but this is a nuisance, because every programme which uses the file must include code to sort out the attributes (which it might not need at all) from the data.

> *Again, I speak from experience. There used to be things called something like "self-describing files", though I can't find a reference. Anyway, I produced an information-retrieval system called Croak which used such files[14], where the structure of the file record (a tree) was described in the first file record. It worked well, but later attempts to reimplement it in somewhat abbreviated form have been plagued by the requirement to deal with the first, rather special, record. As the file is usable for many purposes without the structural details, it has commonly been omitted, so the interesting parts of Croak haven't worked for a very long time. This has not dampened my enthusiasm for the principle; I am even now working on a set of markup files[15] which follow much the same pattern but with differences in detail, and the same sort of record structure information would be useful – but, again, it would be a nuisance in programmes which didn't need it, so it's happened occasionally, but not often. Yes, it's bad programming discipline, but if I had the file structure I wanted it wouldn't be.*

I record for possible (though unlikely) future reference that I speculate on the possible complementary nature of the attributes of programme and data; somehow, programme and data must match if they are to be compatible. (It is not at all clear to me how that can be done except in the most trivial way – "I want a file of 10-byte records"; "I am a file of 10-byte records" – but it would be satisfying if the union of the attribute sets of programme and data (perhaps more than one of each) were in some sense demonstrably sufficient to run.) There is obviously some link between this thought and my obsession with Vocabulary Translation Analysis[16], but I'm not sure what it is.

My table is intended to be illustrative, not exhaustive. I don't think that it is possible to construct an exhaustive table, as there are always other sorts of association one might want. (An exhaustive set of headings might be possible, but I don't know what it is.) I have selected a collection of items reasonably representative of current practice, but twenty years ago I would probably not have included items such as the API or project structure – and the significance of project structure depends on the source of your programming language software.

Likewise, not every item I have listed is essential. A good source of extreme cases is early practice : for example, the name is unnecessary (in the IBM1130 software, at least one programme assumed that it owned the disc and wrote wherever it pleased thereon, using only disc addresses; and in the disc monitor system for the same machine, omission of a name could imply that the source was an area called working storage); you don't need a body (The Unix /dev/null is an example); you don't need attributes (the IBM1130 monitor system again – in fact, there were two 1-bit attributes, but they were almost invisible). These are curiosities, but they serve to establish the principle that what we see now as essential might not be so.

IS IT REAL ?

A reasonable objection to my claim is that my taxonomy is so general that it will fit almost everything that is – at least, everything which can be denoted by a name. The thing, and its name, will always (?) have attributes of some sort and relationships with other things, so there will be ways to fill in the table. Is the taxonomy too general to be useful ?

That is a reasonable objection, but I think it is a matter of vocabulary rather than substance. I chose the terms I used in the table so that they would include both programmes and files, but couldn't find anything more specific than those I've given. I think that my case depends not on the possibility of filling in the blanks in the table but the observation that, when you do so for files and for programmes, the items in corresponding cells are themselves related in significant and interesting ways. For example, both programmes and files have structure, and there are many parallels to be drawn between them.

Or perhaps it is more correct to say that I believe that there *ought* to be many parallels to be drawn between them. The table has turned out to be both more and less convincing than I had expected when I started to write this note. It is less convincing in that the parallels I sought have not turned up as readily as I had hoped; but it is more convincing in that when I have sought parallels I have usually been able to find them, or hints that there are interesting things to find – though not all have had snappy names which I could put in the table.

One example will suffice as illustration, if only because I don't want to spend any more time on this note. I mentioned the structure of programmes and files as an example of duality, but to find the parallels you have to burrow a little. A simple file is a sequence of records all of the same form but with different data in each record; a programme with associated structure is one written in a rather simple machine language. where each item has a operation code and some form of address. A programme in a higher-level language might have much more complex structure, with iteration, condition, sequence, and so on as significant relationships within the structure. We are not accustomed to thinking of such relationships in files, though they have parallels in data structures. How do we store complex data structures ? "With difficulty" is perhaps the only satisfactory reply – but we contrive to store the programmes. We have standard ways of converting highly structured code into overtly purely sequential machine code; why can't we compile our complex data structure into overtly purely sequential files ? – then the parallel would be pretty well complete.

There is obviously a lot more to say about that "argument", but there is something in it. For example, there are problems with the directly-linked databases to which it leads. On the other hand, one might also think of languages such as Lisp, where the distinction between programme and data is deliberately vague.

WHAT I FOUND OUT TOMORROW.

One of the disadvantages of defining "today" as a fixed point is that it precludes new ideas. The heading above is my way of circumventing this barrier so that I can record what has come out of all this mental agitation. I think this is a lot better.

Perhaps the real real answer to the question rather loosely considered in the previous section is that the taxonomy is indeed general, which is why it's useful. The generality is also a very good reason for building corresponding structures into computer files, which are correspondingly general repositories for anything we want to keep in computing terms.

For historical reasons, largely concerned with what I was thinking about today, I've said a lot about programmes. It is gratifying that programmes fit into the pattern, but they are just one sort of thing that we want to keep. We have seen that formatted text is also structured in a related way.

If I had a case, I'd rest it.

REFERENCES.

1 : *Inside Macintosh*, volume 1 (Addison-Wesley, 1985).

2 : G.A. Creak (ed) : *The Zeno system* (Auckland University Computer Centre Technical Report #22, 1984).

3 : G.A. Creak : *Files, editors, and things* (unpublished Working Note AC12, 1979).

4 : G.A. Creak : *Files In Zed and Zeno – positively the Last Exposition* (unpublished Working Note AC18, 1979).

5 : J.E. Sammet : "Basic elements of COBOL 61", *Comm. ACM* **8**, 9-17 (1965) (reprinted in *Programming systems and languages* (ed S. Rosen, McGraw-Hill, 1967), 119-159).

6 : G.A. Creak : *LSI Basic* (unpublished Working Note AC42, 1985).

7 : G. A. Creak : *Student Basic* (Auckland University Computer Centre, 1975 ?).

8 : M.G. Lane, J.D. Mooney : *A practical approach to operating systems* (Boyd and Fraser, 1988).

9 : G. A. Creak and Robert Sheehan : "A Top-Down Operating Systems Course", *Operating Systems Review* **34#3**, 69-80 (July 2000).

10 : D. Kotz, N. Nieuwejaar : "Flexibility and performance of parallel file systems", *Operating Systems Review* **30#2**, 63-73 (April, 1996).

11 : *Turbo Pascal for Windows users guide* (Borland International, 1991), 121-125 (– and many other places, but that's the closest to me at the moment).

12 : D.E. Knuth : "Literate programming", *Computer J.* **27**, 97 (1984).

13 : G.A. Creak : *Preliminary instructions for using Dietician* (unpublished Working Note AC20, 1979).

14 : G.A. Creak : *CROAK : Collected References Organised for Access by Keywords* (Auckland University Computer Centre Technical Report #18, 1980).

15 : G.A. Creak : *Standards for information files* (unpublished Working Note AC130, 2000).

16 : Vocabulary Translation Analysis is not yet documented adequately, but some examples appear in : G.A. Creak : *Analysing WordKeys* (unpublished Working Note AC103, 1996).