WORKING NOTE AC117

Alan Creak
5 December 1997

DESIGNING THE DOCUMENT FACTORY

HISTORY.

In early 1997 I volunteered for the job of information coordinator in the Computer Science department, which I invented for the purpose. (The job, not the department.) This act was regarded as somewhat odd by some of my colleagues, but I had three reasons for my strange behaviour.

First, I had recently returned from leave, and had not yet acquired any administrative responsibilities. While the experience was pleasant, there was no doubt that it wouldn't last, so it was in my interests to set up for myself a comparatively congenial task.

Second, for years I had been growing increasingly impatient with the deplorable state of the department's information system. Information is something about which we are supposed to know, but there was little or no evidence that we did. Information was hard to find, out of date, incomplete, and inconsistent. It seemed to me that at least I couldn't do any worse.

Third, I had some suspicion even then that the task would be bendable in the direction of real-time systems. It is a collection of activities executed by several processors (some human) and (almost ?) all triggered by external events. While there are few really hard deadlines, there are certainly constraints which are significant in the time scales involved, and in many cases accurate sequencing is essential and quite difficult to track. Alaso, much of the task is a matter of shuffling and combining information in various forms, and I had already thought of that as a type of assembly task[1] with analogies to assembly operations in manufacturing systems. I hinted at the connection in an earlier note[2], though I don't remember whether or not I had at that time made the connection with PDL[3]. In any case, the chance of getting some research activity into my administrative job was a strong incentive.

The PDL theme certainly impressed itself upon me as I worked through the first big job which came my way – the assembly of the department's handbooks. It had always been my intention to automate the process as much as possible, and as I carried out all the operations by hand in 1997 this seemed increasingly desirable, and increasingly feasible. Early enthusiasm for PDL was tempered as I realised that the job was more complicated than I had first supposed, but with further acquaintance I began to see it as based on a set of simple unit operations, and the PDL motif reappeared.

WHY PDL ?

One reason for thinking in terms of PDL is, obviously enough, that it's my language and I want to use it. This is not a very good reason. Fortunately, there are better ones. Here is a selection, of uneven quality.

- I have already mentioned that the task is an assembly task, comparable in many ways to that of assembling a product from several component parts. PDL is designed for exactly this purpose.

- PDL is intended for use in a distributed system. The department's information system, should there be one, is certainly distributed. Various sorts of information live on different machines, much of it controlled by different people; the possibility of a distributed system composed of components running on all the machines involved is not silly.

- I would be able to experiment with the design of a "factory" from the very beginning. This is not an opportunity which often comes my way, but it's part of the design process as set out in my plan[3] for PDL, and the experience would test my ideas.

- A suitable medium for the construction of such a system was to hand. Java is also designed as a means of building distributed communicating systems, and it seemed likely that its facilities could be used to good effect in implementing the sort of system envisaged.

WHY NOT PDL ?

There are two main arguments against the use of PDL : that it is totally unknown, and that it might not work. The first does not apply to me; the second is a risk I'll have to take, but I think it's not a great risk.

Natalie Spooner's work on a recent project[4] and in her current master's degree thesis studies is encouraging.

If the worst happens, we'll be no worse off than we are now. Indeed, we shall almost certainly be better off, because even if the programmes don't work the data files which I shall have to collect in order to test the system will remain, and can be used with some other programme (or, in the worst case, by hand) at some later date. These files are the real focus of the information coordinator's job[2].

FIRST SKETCH OF THE FACTORY.

A few preliminary exercises made it clear that it would not be wise to begin by trying to automate the complete handbook exercise at one go. They also came close to persuading me to do it some other way (this is the deterrent phase which I mentioned earlier), but informal explorations of a small selection of other ways turned up just the same problems. I concluded that PDL would be useful if only to give a unifying vocabulary in terms of which all the parts could be described. The real clarification came when it became evident that much of the difficulty was of my own making; I was ignoring my own principles by rushing into system development before designing the plant. A more systematic approach to the same questions following my own advice[3] proved much more satisfactory, and indeed emphasised the resemblance between the handbook exercise and an assembly plant.

The wisdom of starting with a very simple system was underlined by this review, so my first task was to decide on a suitably simple system to implement. Dreams of a simulated factory receded, to be replaced by visions of a garage-scale operation. Well, if Hewlett and Packard can do it[5], so can I. A correspondingly reduced task was also easy to define; there were one or two very regular parts of the handbook assembly which had proved comparatively simple to manage, but also included components which would eventually become significantly more complicated as the system developed. These tasks were, most evidently, the translation of the Courses and People files[6] into the shape in which they would appear in the handbook. I chose to work on the Courses file, partly because it had a little more variety than the People file, but also because it had fields for which more elaborate methods would eventually be required. An example is the **lecturers** field, which in the simple version is taken from the Courses file, but which to ensure accuracy should really be constructed by searching the Functionaries file. Notice that, if the PDL system works, that change should be a matter of simply replacing one "machine" by another.

I therefore begin by defining the object of the exercise. It is interesting that, even to get as far as the modest definition below, I had to redesign the world by deciding on a common format for the course information in the two handbooks. This is one of the less significant symptoms of the disarray in the department's documentation efforts.

> The output of the factory is a file of course details formatted by Word and ready for inclusion in either of the department's handbooks.

Almost all the rest remains to be determined, which is a very unusual point from which to begin a significant plant design. The only precondition is that the raw materials must include the data for presentation, but other details – for example, the form and structure of these data – remain undefined. There is one mild constraint – that the files be usable for other purposes in the department's information systems – but as it's never happened before, and the information systems in question themselves remain to be defined, the constraint is minimal. I shall assume that it is best satisfied by ensuring that the structure of the data is simple and straightforward, which is an assumption which I would have made in any case.

A purist might assert that no such assumption is justified without significant attention to the design of the information systems. In principle, being a purist, I agree; in practice, it isn't quite so simple, for at least two reasons. First, although I would quite like to produce a master plan – and, indeed, proposed that as an important first activity[2] – I haven't time to do it. It has become clear to me that any such information system is likely to be exceedingly complicated, and that it will be more helpful to me to have a tool such as the proposed PDL methods with which I can address problems as they arise than to know exactly what I'm doing right from the beginning. Second, and logically more compelling, a complete design is in principle impossible, because the systems required will inevitably change as facilities, methods, courses,

and regulations change, and much of this inevitable change is outside our control. In such circumstances, the best way to proceed is by developing systems which have as good a chance as possible of adapting to new demands – and sticking to simple structures is as good a way as any of doing this.

Whatever the details, though, it is clear that the input to the factory must be a file containing some representation of the material to be presented in the handbook, and that it must work by selecting material from the input, performing certain editing operations thereon, and presenting the material as output. How should the data file be formed in order to provide the required information, and also to satisfy the criterion of simplicity advanced earlier ?

A good answer is almost certainly that it should be a plain text file, with any necessary structure provided by distinctive text markers. The very persuasive advantage of using such a file is that it can still easily be read and understood even if all software specially designed for using it is lost. It is perhaps the one sort of file which has survived the waxing and waning of fashions in software over the years, while different sorts of word processor and spreadsheet and database have come and gone, incurring significant costs in either conversion or incompatibility at each change. A second advantage is that it is, in effect, infinitely flexible, and, with sensible design from the start, can readily be adapted to accommodate new requirements or desirable changes in the file structure. As I have remarked, changes in the department's information are likely, so the flexibility is worth having. For these reasons, I chose the text file as the underlying file structure :

> The input to the factory is a text file containing the required information, with structure marked by identifiable text strings.

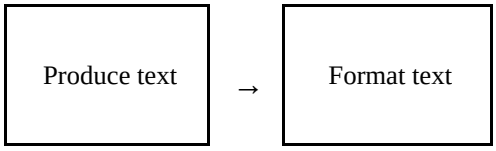
For this first simple exercise, there is one more useful general statement which can be made about the process. It is essentially a repetitive reformatting operation in which a unit of information from the input file is massaged into a corresponding unit of the output file; no complicated distant references in the file are required, and the information units in each case are compact.

> Much of the factory's processing is conducted by converting one record at a time from the input file into the corresponding record for the output file.

Why "much of the processing" ? Why not all of it ? Because I'm looking ahead a bit, and from what I see I can be sure that the statement is unlikely to hold true for very long. All will become clear in due course once we work out a few more details, and we shall now proceed to do just that.

So far, I have kept to generalities, with no mention of any specific means of carrying out the necessary transformations of the text from what is essentially a mark-up form into the result, which is described by the most specific definition I have given – as a formatted Word file. This specificity is reasonable, as the target of the exercise must be precisely defined if we are to devise a means of reaching it. Other constraints are imposed by the context and the practicalities of surrounding operations. For example, the acquisition of the copy for the files is conducted by electronic mail, so it arrives with little or no intrinsic formatting. At some point, the plain text file must be introduced to Word, where various formatting operations are possible. There are many ways in which the various editing operations can be divided between Word and other means, but a convenient guide might be to restrict the Word operations to those which can be carried out with a simple Word macro using a suitable template. The overall plan is then to use simple text processing techniques to sort out the required items from the data files, leaving one complex but manageable final operation for Word. This is the background for my assertion that much of the processing (the text processing) falls into the simple pattern described, but a portion (the Word operations) are carried out by different means.

At the highest level, therefore, the handbook as a whole passes through two stages of construction, each involving an operation on the handbook as a whole :



The first operation is carried out by a text-assembly machine, while in the second a Word machine is used to format the document. There is a close analogy with setting up a component for a machining operation; the first step resembles the careful placing of the workpiece in the workspace of a machine tool, providing such jigs and anchor points as are need for the proper performance of the second step, which is the finishing operation. This analogy emphasises that in order to design the details of the first stage it is necessary to understand the capabilities of the machine used for the second stage.

The major PDL programme is therefore quite simple, executing the first machine and then the second. There is a little more variety than that description suggests, because in fact three sorts of output are required from the same raw material : the undergraduate and graduate handbooks in text form, and a version in HTML for the world-wide web. One way to achieve the desired result is to construct separate – though obviously very similar – special-purpose machines for each handbook version. This is not a very satisfactory solution, though, because each of the versions is likely to change perceptibly from year to year, and a design which requires further PDL programming every year to keep up leaves something to be desired. It is likely to be more satisfactory to make each of the machines programmable; this will cater not only for the year-by-year evolution of the handbooks, but also for the three different versions. Both machines must therefore be made programmable, and will different programmes for the different cases. We are already at the level of flexible manufacturing systems ! Despite this, though, in this discussion I shall not produce a programmable machine; it will be quite enough for the moment to produce any machine at all, and programmability introduces an additional level of abstraction which should not be hard to implement, but might.

The two top-level machines are very different in nature, and I shall therefore discuss them separately. I deal with the text-assembly machine here, now, and certainly; further discussion of the Word machine will be found elsewhere, later, and perhaps.

THE TEXT-ASSEMBLY MACHINE.

The text-assembly machine is complex, because it contains what amounts to an assembly line for the construction of the prepared text to be converted by Word into the handbook. The raw material for the text-assembly machine is mark-up text in the Courses file, and we have seen that it is organised into records corresponding to the courses that must be described in the handbook. The major structure is therefore a basic operation applied to all records. The records are independent and do not interact, so it doesn't matter whether the implementation deals with the records sequentially, in parallel, or whatever. In terms of the factory analogy, the important point is the the same machines can be used for all records, which makes sense of the choice of PDL and the assembly-line model, and this in turn results in a simple course-by-course sequential operation.

Within each course, though, there is no reason why any such orderly behaviour should prevail. In practice, it does for most of the time, but that's an accident; it is simply because the Courses file was originally constructed from a version of the handbook. As time goes by, it is quite likely that the two will diverge significantly. The process should therefore not be seen as a simple sequential editing task. An appropriate design is to perform the conversion in two steps, first fetching the whole record from the Courses file and analysing it into its component fields, then assembling the prepared text file by picking and choosing from the material as required.

How should the primary mark-up files be constructed ? A record from the Courses file is (currently) composed of fields identified by these headers :

- !!number||
- !!title||
- !!prerequisites||
- !!restrictions||
- !!assessment||
- !!where||
- !!when||
- !!lecturers||
- !!tutors||
- !!texts required||
- !!texts recommended||
- !!description||
- !!contents||

Each field might or might not be followed by text. Many of the fields are expected to be occupied (there is always a **number** field and a **title** field), but in any individual record some may be vacant or even missing (not all courses have **restrictions** or **tutors**). This is an important characteristic of the file, because if it were not so it would be difficult to add new fields when they were required without extensive manipulation of the file. For example, suppose at some future date it was found useful to recommend specific items of software for courses; it would then be desirable to define a new field (say, **software**) which could apppear in any record, but if the field was not required in all courses it would be unnecessary to include it in every record.

The mark-up file should therefore begin with a template listing all the permitted fields, with each record following structured as an arbitrary selection from these fields. With the flexibility of optionally appearing fields, defining a new field becomes a simple insertion of the definition into the template, rather than an editing operation affecting all records in the file. It is not accidental that this structure resembles that used in the Croak system[7] for information storage and retrieval; while Croak never swept the world, it worked rather smoothly.

In some cases it is necessary to separate components within a field. A typical example is the **texts required** field, which might contain a list of several books. When printed, the books must be presented on separate lines, so they must be separated in the text. The obvious separator is a new line – but that is not satisfactory in practice, because the revision of material for a new textbook is carried out largely by electronic mail, which can introduce addition newline markers at arbitrary positions. An alternative is an explicit mark-up symbol, but I rejected this for reasons not connected with this design exercise. (During the revision, the material passes out of my control and is changed by my highly intelligent colleagues, a few of whom are quite unable to conform to simple conventions; special markers would probably not be added, and might even disappear or be subtly changed. It is easier to control damage if there is less damageable material.) Real item separators, as distinct from accidental line breaks, are therefore represented by pairs of newline characters.

The output from the text-assembly machine is not a file of this standard format. Instead, it is a string of text prepared for reading by Word. The components of the text-assembly machine are therefore likely to operate by fetching a field from the Courses record, performing some operation on it – probably not a very complex operation, because most of the text is copied fairly directly – and attaching the result to the end of the output file being constructed. The metaphor of a simple assembly operation is good. The ideal system is not an assembly line, though; as we desire it to be programmable, it must be a job shop, where the machines of different types are available to be used without constraint on their order in response to a controller which interprets the programme. As I observed earlier, though, I shall begin by ignoring this requirement in the interests of simplicity, so what I shall produce is an assembly line for building the 1998 versions of the handbooks.

This is the system which I shall now try to design. I shall base my discussion on the scheme described in Chapter 3 of my earlier report[3], not because that is the only way to proceed but because I have never before worked it through with a real example. (My scheme isn't unique, except in that it's mine; others have been described, but most follow generally similar patterns. Kerr[8] reviews several, among other things, in an interesting book.)

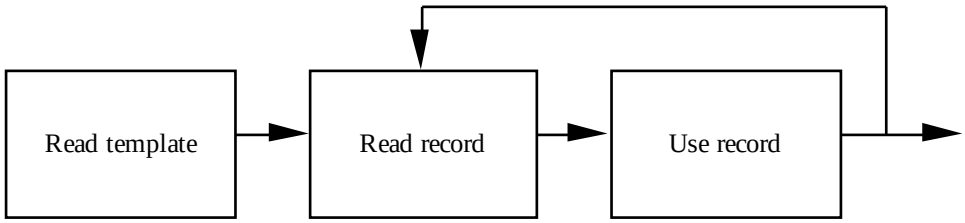
FIRST STEP : THE FABRICATION PROGRAMME AND PLANT DESIGN.

The Fabrication Programme is simply a set of instructions for the assembly. It is not formal, but describes the operations required. It does not specify any particular machinery – but it does imply some machinery, which is why it goes together with plant design. It is comparatively easy to write, because there are as yet

no constraints; the instructions can simply be put down as one might write them for someone else to carry out rather than for a machine.

The raw material is presented to the assembly line in the Courses file as a sequence of records, which (because of the assembly line model) will be treated serially. There are two parts to this operation. First the plant must be set up to deal with the Courses file, and ready to read it record by record as required. Then the records must be read and converted into the required output form. To make the required operations easy, it will be useful if the reading operation takes a complete record from the Courses file and presents it in such a way that its fields are all equally accessible by name. This is reasonably straightforward, because (as described earlier) the file begins with a template which describes the sequence of fields.

The basic structure is therefore :



and the associated fabrication programme can be written in pseudocode as :

```
Open courses file
repeat until end of file
{
     get the next record
     convert into output text
}
```

With this background, it is straightforward to work through the record, producing instructions on the way. There are only a few different sorts of format required.

CONVERTING THE RECORD.

Just what is involved in "convert into output text" ? Here's a specific example to show what is required. This is a real record from the Courses file :

```
!!number||
415.340

!!title||
OPERATING SYSTEMS

!!prerequisites||
(415.210 and 415.231 and 415.232) or (415.212 and 415.233)

!!restrictions||
415.341

!!assessment||

!!where||
City and Tamaki

!!when||
Second semester, 3 lectures per week

!!lecturers||
```

```
Dr Alan Creak (supervisor) (50%)
Robert Sheehan (50%)

!!tutors||

!!texts required||
A.Silberschatz and P.B. Galvin, Operating Systems Concepts
(4th Edition) (Addison-Wesley) (text)

!!texts recommended||

!!description||
A computer’s operating system is the collection of software
which deals with the essential organisational tasks which
must be carried out if the machine is to deliver services to
people who need them. Its responsibilities range from dealing
with the people who use the system, to moment-by-moment
allocation of resources, to managing different activities in
progress. The paper describes the functions which the
operating system must perform, and critically examines ways
in which these requirements can be satisfied.

!!contents||
Why we need operating systems, and how they have developed.
Making the system usable: the computer-human interface.
Making the system safe: protection, security. Managing data:
memory management, file management. Managing computing:
processes, processor management, concurrent processes. Making
the system work: driving devices, scheduling.
```

And here is the corresponding record as it should appear in the handbook :

415.340 ***OPERATING SYSTEMS***

Prerequisites:	(415.210 and 415.231 and 415.232) or (415.212 and 415.233)
Restrictions:	415.341
Assessment:	
Where:	City and Tamaki
When:	Second semester, 3 lectures per week
Lecturers:	Dr Alan Creak (supervisor) (50%) Robert Sheehan (50%)
Texts required:	A.Silberschatz and P.B. Galvin, Operating Systems Concepts(4th Edition) (Addison-Wesley).

Description:
A computer’s operating system is the collection of software which deals with the essential organisational tasks which must be carried out if the machine is to deliver services to people who need them. Its responsibilities range from dealing with the people who use the system, to moment-by-moment allocation of resources, to managing different activities in progress. The paper describes the functions which the operating system must perform, and critically examines ways in which these requirements can be satisfied.

Contents:
Why we need operating systems, and how they have developed. Making the system usable: the computer-human interface. Making the system safe: protection, security. Managing data: memory management, file management. Managing computing: processes, processor management, concurrent processes. Making the system work: driving devices, scheduling.

Now we are ready to consider what is required to convert one into the other.

BITS ABOUT WORD.

I remarked above that, because of the nature of the relationship between the text-assembly and Word machines, it was necessary to understand the details of the operations which could be performed by the Word machine in order to design the text-assembly machine. Now it is time to do so.

It is also time to decide on the division of labour between assembly line and Word, and I shall address this question first, because the answer strongly affects the amount of Word which must be considered in designing the text-assembly machine. The question is significant because there is a great deal of overlap between the capabilities of the two machines. Certainly, both are capable of doing almost all the job; it is quite possible that Word could do it all.

For several reasons I don't propose to let Word do it all. Given that the target is defined as a Word file, I can't avoid some involvement with Word, but here are three reasons for minimising the contribution of Word as far as possible.

Dependency is unsafe : I believe that a good documentation system is important, and I have learnt from experience that depending on proprietary software for continued service over a long term is unwise. Fashions change, and facilities provided with one version of software can disappear from, or be enhanced to the point of uselessness in, the next. It has also been known for software to go away entirely; while this is a bit unlikely for Word, playing safe is not a bad policy. Worst of all, none of these changes is under our control in any way at all, and in the long term even a version which works now will become impossible to run on newer operating systems.

Our wider system : There is more to be considered than the handbooks. This activity must fit in with a much more extensive array of information-processing tasks[2] which are none the less important despite their current non-existence. Many of these tasks are much less likely to be conveniently managed by Word or anything like it, so there will inevitably be more home-grown software, so the handbook development will not be an isolated case.

PDL : From my point of view, the more PDL I manage to fit in, the better. I want to exercise the language and methods as much as I can, and a large connected system seems to be a good way to do it.

Word is dreadful : Constructing Word macros sounds like a good idea, but (writing after the event with the benefit of experience) it isn't. It might be easier if I cared to spend more time learning Word Basic, but I don't, much. In practice, the obvious next steps not uncommonly turn out to be impossible, and you spend a lot of time circumventing obstacles which seem to be unnecessary. The less there is of this, the better.

(In case you think that adds up to four reasons, I should add that I consider the last two to be half reasons.)

With these considerations in mind, then, we return to the question of just what we expect Word to do for us. I think that the very minimum I can get away with, short of trying to construct Word files without using Word, which is silly, is to make Word do the final formatting, and no more.

In this case, the object of the exercise is to produce a sequence of text strings which can be converted easily by the Word machine into the final form required. Inspection suggests that, given the text itself, just two sorts of component are required : a means of identifying regions of text to be formatted with some convenient paragraph style, and a means of identifying smaller regions of text which must be given different local character styles, such as italic or another font. It also seems likely that means for constructing tables will be useful, though I can get away without that for the Courses file, and propose to do so.

A simple example in the introductory note[6] illustrated how this text :

```
!!assessment||<eol>
70% examination, 30% coursework<eol>
```

(where <eol> denotes the end-of-line symbol) could be converted into this :

Assessment : 70% examination, 30% coursework

Simple Word operations are used, and a style must be defined. That conversion is particularly simple, because the whole string to be changed can be expressed as a single paragraph. Multiple paragraphs can be handled too, though in this case it is necessary to use the Word "Use Pattern Matching" replacement method. The text to be changed can then be represented as (for example)

```
!!texts required||*!!
```

which identifies the region stretching from !!texts required|| to the next !!. The same technique can be used to identify local strings which must be given different character styles; for example, markup brackets can be used to identify the strings :

```
This is the !!italicon||Computer Science!!italicoff||
department in the !!italicon||Science!!italicoff|| faculty.
```

and the relevant text items selected using a simple pattern matching operator :

```
!!italicon||*!!italicoff||
```

The required cases are therefore easily covered, and the strings inserted as markers are easy to put in and easy to take out. (In practice, they are not quite so easy to edit, as Word regards exclamation marks as special operators in pattern matching mode, so each "!" above must be replaced by "\!", but it's the principle that's important.)

The notation shown in the example[6] satisfies these requirements for the simple operations needed to construct the courses file. This is not an accident; it was designed for precisely this purpose, but at a stage of development when I thought it would be sufficient to massage the original data files into the final form. The details of the method proposed below are slightly different, but the principle remains the same.

In summary, it is sufficient to mark paragraphs, or ranges of paragraphs, requiring a particular paragraph style with a single leading distinctive mark-up symbol (and obviously to insert tabulating characters where necessary), and to bracket shorter sequences of text requiring special treatment with pairs of mark-up symbols. All the rest can be done within Word by a simple sequence of "Replace All" operations.

MAKING THE WORD INPUT FILE.

I shall now write down a sequence of operations which will (I believe) suffice to convert the mark-up file material into the prepared text file for Word. I remarked that there was no constraint on the language I used for this purpose; nevertheless, it is obviously sensible to choose operation names and forms to be simple and descriptive, and in such a way that similar but not identical operations are seen to be connected. The "programme" below was written with these considerations in mind, and I have avoided the temptation to fiddle with it after writing; what you see is what I originally wrote, admittedly after spending quite a lot of time thinking about the task to be accomplished. I think it has worked pretty well.

1 : The **title** line.

This includes the course number and title. The title is placed at a standard inset which is always used throughout the header section of the entry, so a single style (say, "tabulated") could suffice. Instead, I have used a different style – "courseheader" – for this line, because it does stand in a special position, and might require special formatting. (For example, it might be desirable to control the spacing between the course descriptions in the handbook independently of other

formatting features; if a special style is used for the header, that's easy, but if not significant effort in editing the file might be necessary.) The whole is printed in italic, but that too can be incorporated into the style. This sequence will produce a suitable string :

```
put "||courseheader|"
copy !!number||
put "\t"
copy !!title||
endline
```

Here and below, I use "put" to add known strings to the prepared text string, and "copy" to copy fields from the Courses record. "\t" denotes a tabulation character; "\n" will later denote a newline character. The style itself does not appear in this programme; it is used in the Word macro, which includes an instruction of the form "replace all occurrences of ||courseheader| by nothing using style courseheader". (No, it isn't quite as simple as that, but it comes close.)

2 : The **prerequisites** line.

This time, the style "tabulated" is used, as it is with many of the other fields which follow. Notice that in this case only a part of the text must be set in italic, so additional markers to effect this conversion must be inserted. These instructions provide the required string :

```
put "||tabulated|||italicon|Prerequisites||italicoff|\t"
copy !!prerequisites||
endline
```

The **assessment**, **where**, and **when** lines follow the same pattern; they must always be present, they are composed of a single line, and they are formatted in the same way. In Word, the paragraph style is set as for the **title** line using ||tabulated|, then the character style of the bracketed string is set.

3 : The **restrictions** line.

The only difference between **restrictions** and **prerequisites** is that there might be no restrictions. In that case, either the !!restrictions|| record is empty, or it is omitted. Either is possible; it depends on the details of the reviewing process, and what the supervisor of the course has decided to do with the material presented. A suitable instruction sequence is :

```
if !!restrictions|| is not empty
{
     put "||tabulated|||italicon|Restrictions||italicoff|\t"
     copy !!restrictions||
     endline
}
```

The fields for **tutors**, **texts required** , and **texts recommended** are similarly optional, so they share the conditional nature of the **restrictions** line, but in their formatting requirements resemble the **lecturers** field.

4 : The **lecturers** field.

The new feature of the **lecturers** field is the possible requirement for a format with several lines. The line breaks are encoded in the text file within the **lecturers** field. An alternative would be to have a number of **lecturers** fields, one for each lecturer concerned, but it seemed likely that this would be more easily garbled in the handbook revision process. In either case, some sort of iterative structure will be necessary in the assembly instructions; here is a possible form :

```
scanning !!lecturers||
{
     put "||tabulated|||italicon|Lecturers||italicoff|\t"
     copy !!lecturers|| until "\n\n"
     while !!lecturers|| is not ended
     {
          endline
          put "\t"
          copy !!lecturers|| to "\n\n"
     }
     endline
}
```

Eagle-eyed readers will observe that this won't work with the example Courses record shown earlier, where I have not followed my own prescription by separating the two lecturers' names by a double newline. There is a reason for the inconsistency : I do not even trust my highly intelligent colleagues to manage the double newline consistently. My guess is that they might do so in cases of book titles or paragraphs in the descriptive parts, where they can see the point of the clear separation, but as a list of names is obviously a list of names they are less likely to conform with the text for the **lecturers** and **tutors** fields. As the machines which handle these fields are specialised for that purpose, I can if I wish make them clever enough to deal with an arbitrary number of newlines, but in writing the programme I've assumed that there will be two in the interests of simplicity.

5 : The **description** field.

The **description** field is of a different pattern, which it shares with the **contents** field. The final formatting can be managed using Word in the same way as for the other fields, but the requirements for text manipulations are more demanding. The field consists of an arbitrary amount of text, and is not indented in the tabular form of the earlier fields. Also, the text is organised into paragraphs, but the raw material is quite likely to contain end-of-line characters from its electronic mail transfers.

Perhaps the obvious implementation of this operation is in terms of elementary character-by-character operations : find the next character; if it's end-of-line then check the one after; if that's end-of-line as well, ... I didn't want to use instructions at that level in the specification, because so far I've been able to express everything in fairly consistent string-processing terms, and to introduce a lower level of description would make for more difficult implementation. It would also make it harder to check that the instructions properly implement the required operation.

(In terms of the assembly line analogy, it's rather like having a sequence ... use lathe, use milling machine, man with file deals with rough edges, use milling machine ... Sometimes it's necessary to incorporate men with files, or their equivalent, but the change in level of description makes for awkwardness, and some phrase like "perform chamfering operation" would at least look better. Then in the next stage you can work out how to do it, and might find that you need a bit more in there than a man with a file.)

The instructions below therefore split the operation into a sequence of string operations on the field before it is copied into the text file. This is of some interest, because it can be seen as an analogue of a sub-assembly in a mechanical system; in principle, it could be operated in parallel with the main assembly as a subsidiary "assembly line".

```
copy !!description|| into descriptiontext
in descriptiontext changeall "\n\n" to "||"
in descriptiontext changeall "\n" to " "
in descriptiontext changeall "  " to " "
in descriptiontext changeall "||" to "\n"

put "||italictext|Description"
endline
put descriptiontext
endline
```

The fabrication programme is now complete, and is presented below. In composing it I have made many assumptions, which together amount to an assumption that all the operations I have specified can be accomplished. The next step is to justify this assumption by designing machinery with which the operations can be carried out.

THE FABRICATION PROGRAMME.

The complete fabrication programme constructed from the considerations set out above is shown below. Each segment of the output text is composed with the instructions discussed, or with some combination of them as noted.

This programme was written before any version of the handbook builder was implemented, and is not guaranteed correct. Nevertheless, I am sufficiently confident to assert that something quite like it will perform the required task.

```
--- Header line
put "||courseheader|"
copy !!number||
put "\t"
copy !!title||
endline

--- Prerequisites
put "||tabulated|||italicon|Prerequisites||italicoff|\t"
copy !!prerequisites||
endline

--- Restrictions
if !!restrictions|| is not empty
{
     put "||tabulated|||italicon|Restrictions||italicoff|\t"
     copy !!restrictions||
     endline
}

--- Assessment
put "||tabulated|||italicon|Assessment||italicoff|\t"
copy !!assessment||
endline

--- Where
put "||tabulated|||italicon|Where||italicoff|\t"
copy !!where||
endline

--- When
put "||tabulated|||italicon|When||italicoff|\t"
copy !!when||
endline
```

```
--- Lecturers
scanning !!lecturers||
{
     put "||tabulated|||italicon|Lecturers||italicoff|\t"
     copy !!lecturers|| until "\n\n"
     while !!lecturers|| is not ended
     {
          endline
          put "\t"
          copy !!lecturers|| until "\n\n"
     }
     endline
}

--- Tutors
if !!tutors|| is not empty
{
     scanning !!tutors||
     {
          put "||tabulated|||italicon|Tutors||italicoff|\t"
          copy !!tutors|| until "\n\n"
          while !!tutors|| is not ended
          {
               endline
               put "\t"
               copy !!tutors|| until "\n\n"
          }
          endline
     }
}

--- Texts required
if !!texts required|| is not empty
{
     scanning !!texts required||
     {
          put "||tabulated|||italicon|Texts
required||italicoff|\t"
          copy !!texts required|| until "\n\n"
          while !!texts required|| is not ended
          {
               endline
               put "\t"
               copy !!texts required|| until "\n\n"
          }
          endline
     }
}

--- Texts recommended
if !!texts recommended|| is not empty
{
     scanning !!texts recommended||
     {
          put "||tabulated|||italicon|Texts
recommended||italicoff|\t"
          copy !!texts recommended|| until "\n\n"
          while !!texts recommended|| is not ended
          {
```

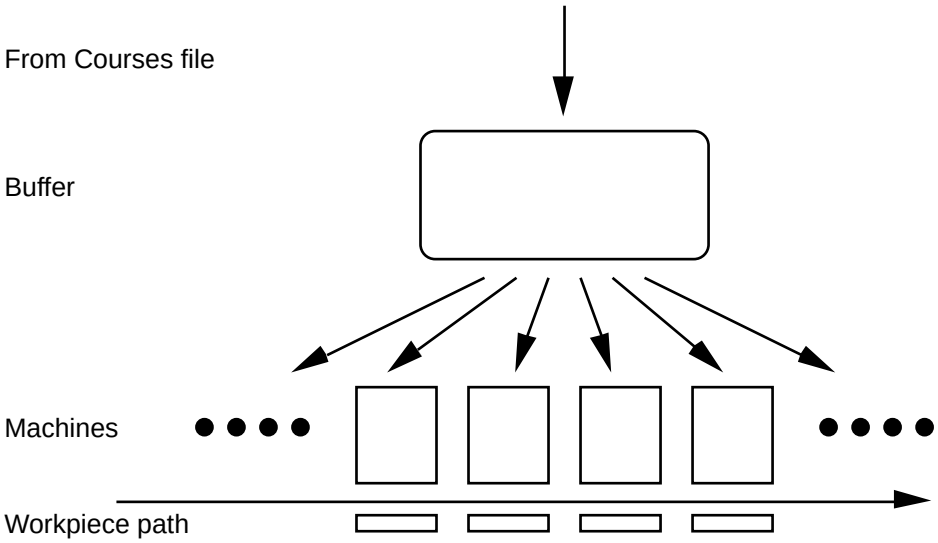
```
                endline
                put "\t"
                copy !!texts recommended|| until "\n\n"
            }
            endline
        }
    }

    --- Description
    copy !!description|| into descriptiontext
    in descriptiontext changeall "\n\n" to "||"
    in descriptiontext changeall "\n" to " "
    in descriptiontext changeall "  " to " "
    in descriptiontext changeall "||" to "\n"

    put "||italictext|Description"
    endline
    put descriptiontext
    endline

    --- Contents
    copy !!contents|| into contentstext
    in contentstext changeall "\n\n" to "||"
    in contentstext changeall "\n" to " "
    in contentstext changeall "  " to " "
    in contentstext changeall "||" to "\n"

    put "||italictext|Contents"
    endline
    put contentstext
    endline
```

SECOND STEP : DESIGNING THE PRODUCTION LINE.

It is clear enough from the start of this design that the task is possible. That's because I have not been silly in writing down the fabrication programme; I am well acquainted with the sorts of operation which make sense when a computer is manipulating text, and I have taken care to use operations which I know can be implemented without undue difficulty. It is also clear that the job can be done in many different ways, but the vocabulary I have used in the fabrication programme suggests that only a rather small number of elementary operations is required.

I have already made one decision in choosing to construct (metaphorically speaking) a production line. The sequential processing characteristic of this production discipline is very simple and correspondingly easy to control; the workpiece being assembled simply moves from one machine to the next, undergoing operations in a predefined sequence. The only flexibility in such a production line (assuming that the machines themselves are not programmable) lies in whether or not individual machines are operated as the workpiece passes by, but that is sufficient to manage the variety required to assemble the courses text.

Just how does the production line model fit the problem of constructing the handbook ? Consider this diagram :



The workpiece is the prepared text record being constructed; it receives the attention of each of the machines in the production line in turn, while they draw their raw material from the Courses file. The production line model is slightly strained at the next level up, for because all the operations in the workpiece simply attach more text to the end, we can construct the complete prepared text file by reusing the completed workpiece from the previous cycle as the initial workpiece in the next. I can't think of a realistic physical analogue of that, but it doesn't change the principle.

A difficulty in constructing a real production line is the expense. If the same operation has to be carried out ten times, it is necessary to install ten identical machines at appropriate positions along the line. With a job shop, machines can be used in arbitrary order, so in principle only a single machine is necessary – but at the cost of much more elaborate materials handling procedures. With the advantage of constructing an abstract machine where expense is not an object, I have chosen the simpler organisation. For the purposes of PDL, the multiplicity of identical machines turns out to be a valuable feature, as it emphasises the distinction between physical and logical machines discussed in the report[3].

What sorts of machine do we require ? Sorting through the instructions, we find several types, which fall into three groups. As the machines are identified, they are named for future reference; the names are bracketed [Thus].

Putters : There is a group of operations in which explicitly identified text is simply placed in the output stream :

```
put "||courseheader|"
put descriptiontext
endline
```

These operations differ only in the source of the text to be copied. In the first instance, it is a literal which is presented in the instruction [PutC], in the second a local variable containing the text is named [PutV], and the third is essentially equivalent to the first, but the use of a distinct form makes it easy to change the end-of-line text if need be [PutE].

Copiers : Three further operations also move text, but they take the material from the current Courses record :

```
copy !!number||
copy !!description|| into descriptiontext
copy !!lecturers|| until "\n\n"
```

The first is a simple copy of a field of text from Courses file to output stream [CopyF], and the second is similar but a local variable is identified as the destination [CopyL]. The third is rather more complex; the absence of a destination implies that material is copied to the output stream, but the copying is to proceed only until the string "\n\n" (two successive line ends) is found in the

input field [CopyU]. It is clear from the usage (and anyway I know because I wrote it) that the instruction can be issued again, when copying resumes after the "\n\n". The end of the input field also ends the copying operation.

Changer : The other operation is used to match a given pattern in an implied string, and to replace every occurrence with a second pattern [Chan] :

```
changeall "\n\n" to "||"
```

The grouping involves no deep analysis; it is an obvious classification based as much as anything on my previous experience of string operations in computers – much as an engineer might design a production line using previous experience of production lines, and knowledge of what sorts of machine tool were available.

That analogy is illuminating, but not perfect. I begin from a position rather less well defined than does the engineer, because I don't have a catalogue of available machine tools. Instead, I have the added luxury (and hard work) of making my own. There are two rather obvious ways to proceed. On the one hand, I can regard each of the operations as performed by a quite different tool, and implement seven separate machines. On the other hand, I can take account of the grouping, which suggests common features in the construction of the tools, and instead aim to produce three varieties of a basic putter and a basic copier, and a single changer. Notions of object-oriented implementations come immediately to mind; more generally, this is an example of the object-oriented classification emphasised particularly by Adrian Krzyzewski in his M.Sc. thesis[9].

Inventing the machinery to implement the operations directly associated with constructing the output text does not exhaust the information embodied in the instructions. There remain a number of fragments associated more with control rather than processing. These are :

```
if !!restrictions|| is not empty
while !!lecturers|| is not ended
{ ... }

scanning !!lecturers||
in descriptiontext
```

I have classified them into two groups, with the first containing instructions which in some sense control the execution of other instructions, and the second containing qualifiers which in some way modify the execution of other instructions.

The `if` and `while` instructions of the first group, the controlling instructions, both manage the performance of a group of machines defined by the brackets which appear as the third member of the group. So far as the main production line is concerned, the effect is to present the groups as single machines which, when operated, look after all the details of what happens within. They are conveniently modelled as subsidiary controllers directed by the main controller, but themselves taking decisions and directing the operations of their own machines. We have invented workcells – or, perhaps more significant, the workcells have invented themselves, for they have turned up as a consequence of a quite natural description of the system.

Another feature appears : the association of attributes with the field names. Two attributes – `empty` and `absent` – depend on the state of the Courses file, while the third – `ended` – depends on the state of the computation. Machinery to test and perhaps (in the case of `ended`) to set the attributes' values must be provided, so we shall invent two further machine types :

Recordtesters determine the value of a particular attribute of a particular field of the current record, and return its value [TestR];

Recordsetters perform the complementary operation of setting the value of an attribute of a field of the current record [SetR].

I include "Record" in the names because it seems likely that at least some of the attributes (`empty` and `ended`, for example) might be useful attached to ordinary variables too if the scanning operations are generalised.

The `scanning` operator has something to do with setting up the **lecturers** field of the input record to be scanned from the beginning, so we might suppose that it sets some sort of pointer to the beginning of the record, and set the `ended` attribute to false. The significance of the brackets associated with `scanning` in the programme is not so clear; though it seemed clear to me as I wrote the instructions that the "scanning" attribute should be associated with the bracketed text, the scope is, in effect, determined by the contained `while` instruction. Should the field be labelled as "being scanned" within the parentheses ? What good would it do ? I don't know (though I can make a guess; see below) – so for the time being I shall assume that the operator only sets the `ended` attribute to false.

The `in` operator is much easier to dissect; it acts as a parameter channel for `changeall`. Though it is in a curious place for a parameter, it is clear that the `changeall` instructions could just as well be written

```
changeall( descriptiontext, "\n\n", "||" )
```

In many programming languages such an instruction would occasion no surprise at all.

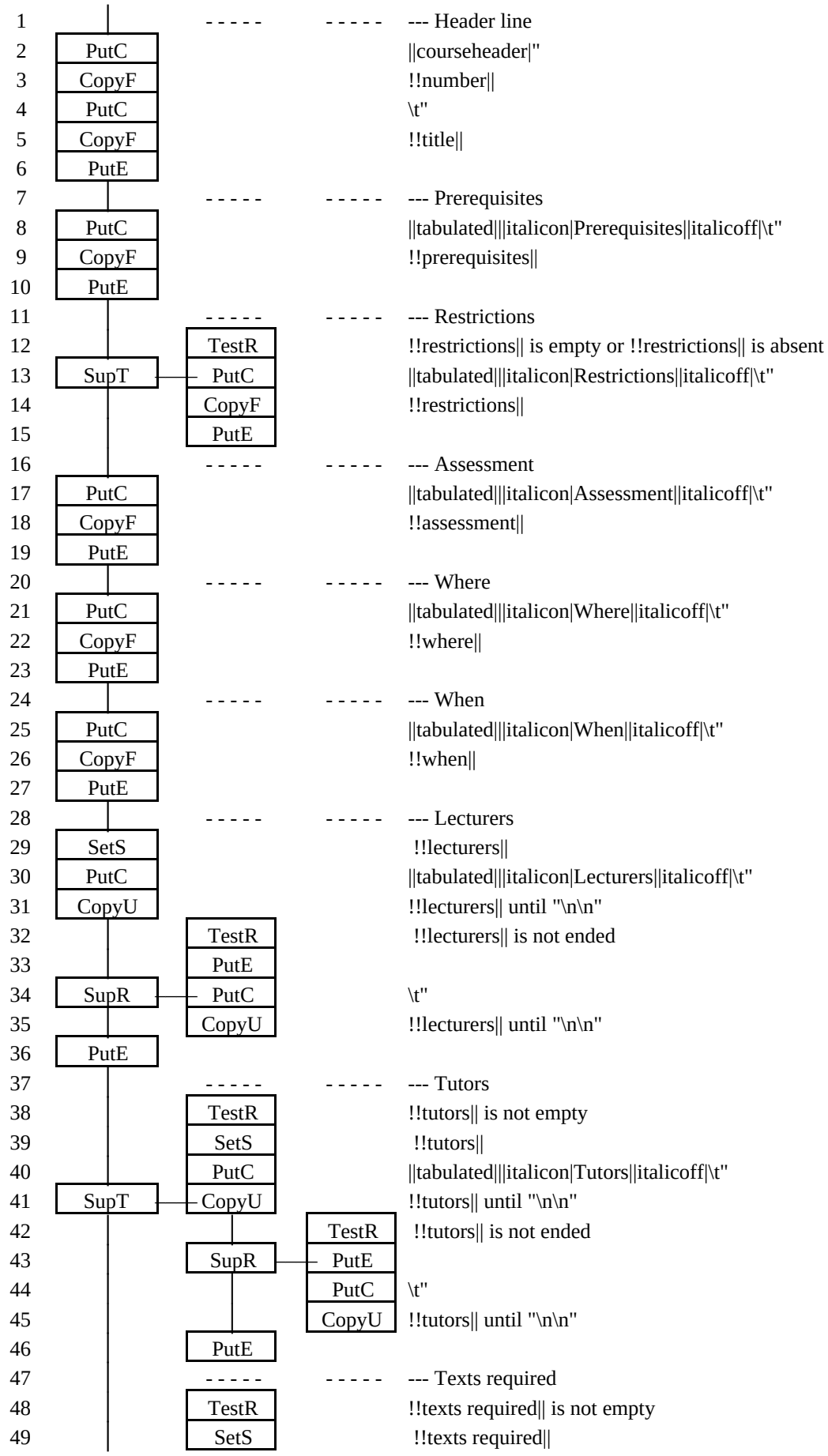
Before leaving these less obvious instruction components, it is interesting to notice that both `scanning` and `in` can be brought into one pattern and given a consistent interpretation. Consider these slightly rewritten versions of the two parts of the programme in which they appear :

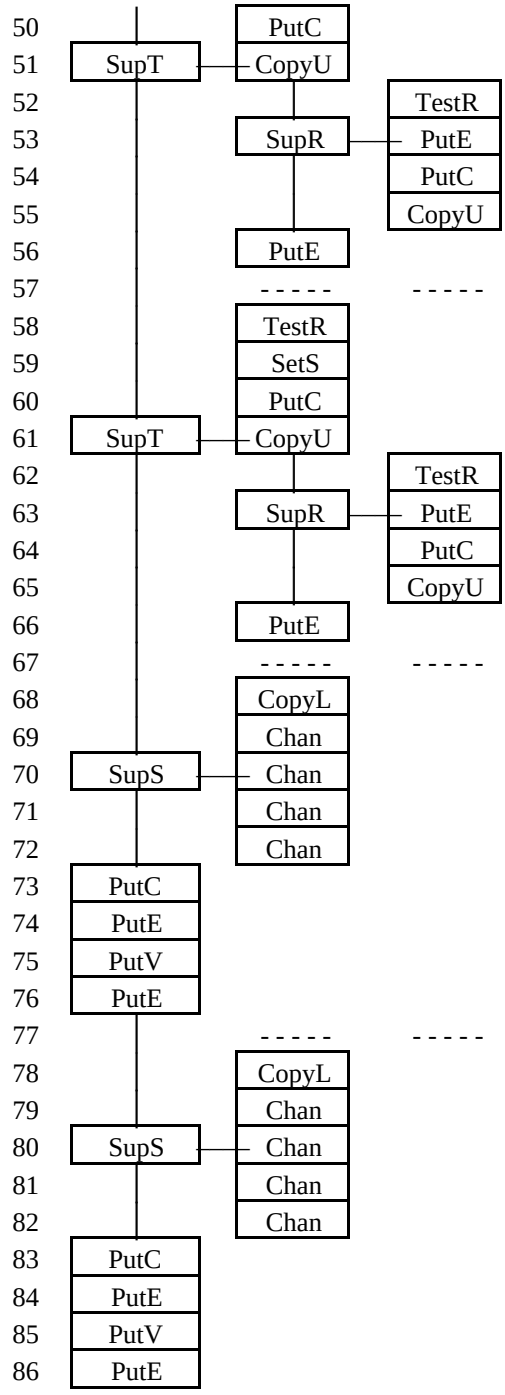
```
scanning !!lecturers||
{
     put "||tabulated|||italicon|Lecturers||italicoff|\t"
     copy until "\n\n"
     while not ended
     {
          endline
          put "\t"
          copy until "\n\n"
     }
     endline
}

in descriptiontext
{
     changeall "\n\n" to "||"
     changeall "\n" to " "
     changeall "  " to " "
     changeall "||" to "\n"
}
```

From this point of view, one could regard the two operators as providing a context item which can be assumed within the associated brackets – a field from the Courses record for `scanning` and a working variable for `in`. Then, within the brackets, items requiring such a parameter but without one specified can use the item of the matching type from the context. One could compare the Pascal `with` operator. The second example is also of interest in that it reflects the "subsidiary assembly line" which I mentioned earlier when discussing the assembly of the text for the **description** field; in this case the different context has a direct counterpart in the model of the process.

What does the assembly line look like when all these machines are connected ? Here's a sketch, derived from the programme above in a rather simple-minded way. It might be bigger than you expected; it is certainly bigger than I expected, but fortunately it is to be built of software. In this connection, it is worth remarking that I have put exactly no effort into optimising it in any way, and, within the scope of

the current exercise, I don't propose to; if this works out satisfactorily, there will be opportunity to think of optimising, but for the moment my primary interest is to get it going correctly.





Some comments are appropriate.

Supervisors : A new species of machine has appeared : the supervisor. These come in three subspecies, and are introduced for two reasons.

First, the testing and repeating supervisors, [SupT] and [SupR] respectively, are used to maintain the principle that machines in an assembly line are operated in strict sequence. To do so in a plant where conditional and repetitive execution of some machines is required, it is necessary to employ something like a supervisory controller. Testing and repeating supervisors perform the required functions; they are almost identical in function, differing only in the repetitive execution in one case.

Second, the subassembly supervisor [SupS] is used as a subroutine abstraction device. It appears twice in the assembly line, and in neither case is it strictly necessary. I have used it to mark the subsidiary assembly lines I mentioned earlier – and perhaps as an indication of future possibilities for parallel execution. One further subassembly supervisor could be introduced to

supervise the whole assembly line, giving a useful abstraction for the whole plant. I have not drawn it on the diagram above, but it appears on the much simpler high-level diagram we saw earlier as the "Use record" box. I shall discuss this further when deciding the nature of the control system.

No RecordSetter : As it has turned out, I have not used a recordsetter machine. Instead, I have assumed that the recordtesters will be able to determine their results by inspecting the state of affairs directly, rather than referring to a explicitly recorded attribute. This was not a deliberate design decision, and it might well be that it would be preferable to restore the field attributes as a more standardised technique for determining field properties.

No error detection : The lack of any explicit error detection is the most serious defect of this design. There is some excuse; unless inspector machines are introduced to carry out testing procedures (and it is difficult to see what they might do in this abstract example), all testing must be associated with the machines of the production line, and so far these machines have not been defined in any detail. In fact, I shall not define any error handling facilities because I can't think of anything that could usefully be checked without going to more elaboration than I want at this stage.

This is because all operations are directly driven by the incoming record from the Courses file. Because of the way it is defined, the record cannot be inconsistent in any way detectable by simple analysis, and the operations on the record must be possible in all cases. The worst that can happen is that items in the output file will be empty or garbled, but in either case the error is a consequence of errors in the input file, not errors in the machinery. I do not offer a formal proof of these assertions, but I conjecture that with careful design and analysis such a proof could be constructed.

More significant errors might appear later. For example, the **lecturers** and **tutors** fields will eventually not be copied from the Courses file, but filled from a different source – the Functionaries file. If it should be found that no lecturer is defined for some course which appears in the Courses file, then at least a report should be issued, and some remedial action taken. Before we get that far, though, many other changes will be necessary, and a complete redesign will be unavoidable; at that point it will be important to introduce fault management from the start.

In terms of the design process as described in my report[3], the diagram combines the Line Design (in part) and Controller Configuration databases. It shows nine production lines, each as a connected sequence of machines in one of the vertical columns, but not all of these are different. For example, the sequence running from machines 68 to 72 in the second column is identical with that running from machines 78 to 82, and these are in turn components of longer identical sequences from 68 to 76 and from 78 to 86 in the first column. Because of this identity, it would be possible to design each sequence just once rather than twice (which in fact is exactly what I did, though I noticed the identity informally before the design rather than as a consequence of the design), therefore using only a single description in the Line Design database.

One could also use the same programme in each of the supervisors at machines 70 and 80 in the first column. This can be seen as an example of reuse of code, which is very trendy. With a production line architecture, it is not possible to reuse the same machines, so the operations performed by the two supervisors must eventually be directed to different real machines; that is the function of the Controller Configuration database, one of which is associated with each supervisor. The individual real machines are catalogued in the Machine database[3], in which each machine is identified by type, physical location, and other properties unique to the physical machines themselves.

DEFINING THE MACHINES.

Now we know a fair bit about the machines we need to build the assembly line, so it is appropriate to formalise the decisions by constructing the Machine Type database[3]. This database includes information about each type of machine. Any useful information about the type of machine should appear in the database; in this case, the abstractness of the machines limits the sorts of information which make sense, so this version lists only the instructions accepted by each machine and its subsequent actions. The actions include both operations which contribute to the manufacturing process, and messages produced by the machine.

Machine type :	Instruction	Operation	Response
Putter, literal	"Put <text string>"	The text string matching <text string> is appended to the product string.	"Done"
Putter, variable	"Put <variable>"	The value of the variable is appended to the product string.	"Done"
Putter, end-of-line	"Put"	An end-of-line marker is appended to the product string.	"Done"
Copier, field	"Copy <field>"	The contents of the named field in the current Courses record are appended to the product string.	"Done"
Copier, field to variable	"Copy <field> into <variable>"	The contents of the named field in the current Courses record are appended to the text string in the variable.	"Done"
Copier, field with delimiter	"Copy <field> until <delimiter>"	The contents of the named field in the current Courses record from the current field pointer to (but not including) the next occurrence of the text string matching <delimiter>, or the end of the field, are appended to the product string. The pointer is advanced beyond the <delimiter>.	"Done"
Changer	"Change <text string 1> to <text string 2> in <variable>"	Every instance of text string 1 found in the named variable is replaced by text string 2.	"Done"
Recordtester	"Test <attribute> in <field>"	The value of the named attribute of the named field is returned.	"Found <attribute value>"
		The field is not found.	"Found no field"
Supervisor, testing	"Run"	Runs the machines in a production line if an initial test is favourable.	"Done"
Supervisor, repeating	"Run"	Runs the machines in a production line repeatedly until an initial test is favourable.	"Done"
Supervisor, subassembly	"Run"	Runs the machines in a production line.	"Done"

For present purposes, these machines will be regarded as primitive objects. It will, obviously, be necessary to compose programmes which perform the functions described, but these are elementary. Even so, in a full PDL treatment we would write PDL programmes which describe the tasks performed by the machines. These will not be directly executed by anything as intelligent and programmable as a computer; typically, they will simply describe the only action, or the one or two actions, which the machine is capable of performing. These programmes are intended for the intelligent part of the system when diagnosing faults. As an example, the programme for a literal putter might look like this :

```
Trigger

     Receive Putmessage
          :    from <machine1>.

Procedure

     Append textstring
          :    to Globalstring
     Send Donemessage
          :    to <machine2>.
```

The two messages concerned are defined in the Machine Type database; in the programme they might appear thus :

```
Messages
     Input
          Putmessage    :    "Put <textstring>"
     Output
          Donemessage : "Done"
```

The value of `textstring` is obtained by matching the received message to the pattern given in the message declaration. The two `<machine?>`s mentioned in the programme must be defined whenever a literal putter is built into the production line. They are determined by the Line Design and Controller Configuration databases; in this example, they will both be the putter's supervisor, but an alternative is discussed below. It is assumed that the putter will be able to find `Globalstring` for itself.

THIRD STEP : THE CONTROLLER PROGRAMMES – COMMUNICATIONS.

Programming the supervisors is an altogether more demanding task. These are, by definition, programmable devices, which in a real production line would be computers or programmable logic controllers. They must be provided with real programmes which determine how they react to signals received from any source, and before we can write the programmes we must know just what the supervisors are supposed to do, and that in turn depends on the nature of the communications between the machines in the production line.

This remains to be decided. The links between machines in the production line diagrams show only the sequence of operations, and, by implication, the flow of (somewhat abstract) material. The communication network is quite independent, except that it must satisfy certain constraints imposed by the requirement that the machines must be made to operate in the order determined.

Fortunately, we know something about how this must be done. I shall take the text preparation production line as my example, for it covers all that's needed for the much simpler high-level production line. From the Machine Type database, and the production line diagram, we know that this sequence of events must be caused to happen :

1 : To start the process, send "Put ||courseheader|" to machine 2 (PutC);
2 : PutC.2 will reply "Done" when it has finished;
3 : Then send "Copy !!number||" to machine 3 (CopyF);

.... and so on, for a very long time. (The numbers are the line numbers from the production line diagram.) Provided that this sequence of messages can be provided, the system will work – and, if simply making it work is the only criterion to be satisfied, it doesn't matter how the messages are produced.

For example, we could link the machines directly. We would need a little box between machines 2 and 3 which, on receiving "Done" from machine 2, would send "Copy !!number||" to machine 3, and we would need similar boxes between all adjacent pairs of machines. Alternatively, seeing that in this case we build our own machines, we could make them so that they would operate on receipt of "Done", which would save the translation. As each machine always does exactly the same job, that would be a satisfactory solution. This direct machine-to-machine connection is what I have called[3] the flat system organisation.

The flat organisation works, and has the great advantage of simplicity. For this exercise, where nothing can go wrong (or, at least, where we have no means of detecting errors, which comes to much the same thing), it would be satisfactory. Its major defect is that there is no place in the system where the state of the production line as a whole is visible. Such a higher-level view becomes useful when error management is required, and also when the production line is used to construct several products at once, each in a different stage of development. Without the global view, the nature of the error can be harder to determine, and it becomes very difficult to coordinate the motion of different products through the process so that they don't interfere with each other.

For this reason, I have chosen instead to use a hierarchical model, where each machine receives its instructions from, and reports back to, a supervisory controller. Each controller is aware of the state of the part of the assembly line under its control, and oversees the proper progress of products through the many steps of assembly. Its programme is therefore much more complicated than the example given above for a simple machine; this is the Controller Programme, which is regarded as one of the system databases[3]. Each actual controller has a controller programme, which is determined by the controller type and by the plant which it controls. (That allows for the possibility that controllers of different types might be used to control the same plant – for example, one might wish to replace a programmable logic controller by a small computer.)

As well as contributing to the controller programme, the decision about the communication topology affects the Line Design database. The communications structure must therefore be recorded in this database with the physical relationships between machines.

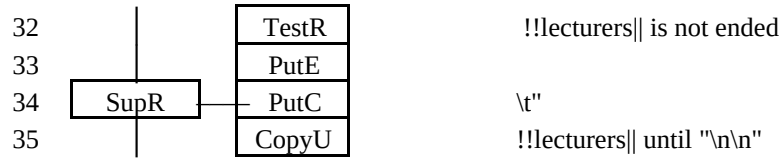
FOURTH STEP : THE CONTROLLER PROGRAMMES – PROGRAMMING.

Constructing the controller programmes is now a matter of determining what each controller must do in order to cause the machines under its control to behave in a way which will implement the fabrication programme using the plant which we have now designed. If the design is correct, then it should be possible to perform this step automatically; I shall not do that, because it would undoubtedly be harder to write the software for the automatic generation than to write the controller programmes by hand. Writing the automatic software would also be much more interesting, but we do want the handbooks next year.

To write the programme for a single controller, it is necessary to identify all the operation sequences which it might be required to control at any time. The signals which initiate the actions must then be identified as triggers, whereupon the programming is fairly straightforward.

In the Courses assembly line, the problem is very simple indeed, because all the controllers (the supervisors) have only one, very straightforward, thing to do, and there is no error management to complicate the issue. As we have already seen, three sorts of supervisor are useful. A subassembly supervisor is triggered by the completion of the previous machine's operation (strictly, by receipt of a message from its own supervisor), and oversees the sequential performance of its production line by alternately sending the appropriate starting messages to the next machine and waiting for its completion messages. A testing supervisor is a little more complicated in that its first machine performs a test, and the remainder of the machines are, or are not, executed depending on the result of the test. A repeating supervisor is slightly more complicated again, in that it also incorporates a loop; I shall describe one such in more detail.

The example will be the first repeating supervisor in the production line. It controls the construction of the text for the **lecturers** field. Here is the segment of the production line :



The corresponding segment of the fabrication programme is :

```
while !!lecturers|| is not ended
{
     endline
     put "\t"
     copy !!lecturers|| to "\n\n"
}
```

The controller programme will be something like this :

```
Trigger

     Receive Startmessage
          :    from UseRecord.

Procedure

     Repeat :
          Send Testrequest
               :    to TestR:32.
          Receive Testreply
               :    from TestR:32.
          SetNormalState
               :    from
                    [
                    endvalue = "no field"   :    Finish
                    endvalue = "yes"        :    Finish
                    endvalue = "no"         :    <>
                    ]
          Send Eolrequest
               :    to PutE:33.
          Receive Usualreply
               :    from PutE:33.
          Send Puttabrequest
               :    to PutC:34.
          Receive Usualreply
               :    from PutC:34.
          Send CopyLrequest
               :    to CopyU:35.
          Receive Usualreply
               :    from CopyU:35.
          SetNormalState
               :    to Repeat.
     Finish :
          Send Endmessage
               :    to UseRecord
```

(In that programme I have used syntax for SetNormalState following a recent proposal[10].) To augment the programme, the Machine Type database must contain these specifications :

```
Output messages :
     Testrequest   :    "Test ended in !!lecturers||"
     Eolrequest    :    "Put"
     Puttabrequest :    "Put \t"
     CopyLrequest  :    "Copy !!lecturers|| until \n\n"
     Endmessage    :    <as defined for UseRecord>

Input messages :
     Startmessage  :    <as defined for UseRecord>
     Testreply     :    "Found <endvalue>"
     Usualreply    :    "Done"
```

- and that completes the most complex of the supervisor programmes. The programme for UseRecord will certainly be the longest, but it is a simple sequence of Send and Receive instructions.

FIFTH STEP : DRIVING THE MACHINES.

- and that completes the most complex of the supervisor programmes. The programme for UseRecord (the overall production line supervisor) will certainly be the longest, but it is a simple sequence of Send and Receive instructions.

FIFTH STEP : DRIVING THE MACHINES.

The story is not yet complete, for (leaving out of consideration the non-trivial question of writing the code to simulate the machines themselves) there are a few details of execution behaviour which can usefully be discussed in this descriptive account of the system. Perhaps the best general account of the execution method is in my note[11] misleadingly entitled "Clarification", but this example gives an opportunity for a few more specific descriptive comments.

The particular feature of interest is the communication between the machines, and particularly how the supervisors deal with this communication. (So far as PDL is concerned, the other machines hardly exist; it is easiest to think of them as essentially stupid hardware entities, responding simple-mindedly to very simple instructions.) The important questions to be settled are how the messages are constructed, sent, received, and interpreted.

For construction and interpretation, the main principle is pattern-matching. In this example, the messages sent are generally known precisely before the event, so they can all be given as string constants, and in most cases the same is true of the messages received. In other cases, declared messages include variables, which can be set before transmission of an output message, and are set by the pattern matcher on receipt of an incoming message. An incoming message is first compared with the declared input messages, and – all being well – matched. What happens in the case of several matches is not defined, as, once again, we cannot predetermine the behaviour of hardware machines which might be included in the system. In a controller, it is likely to be possible to ensure that the combination *{ source of message, message }* is unique; if not, someone must decide what to do about it. (An obvious solution is to throw away the machine which gives identical messages for different faults, but sometimes it is possible to infer more information from the context in which the message was received.) If a received message does not match any declared message, some sort of error has occurred, and should be dealt with by the PDL system, if there is one.

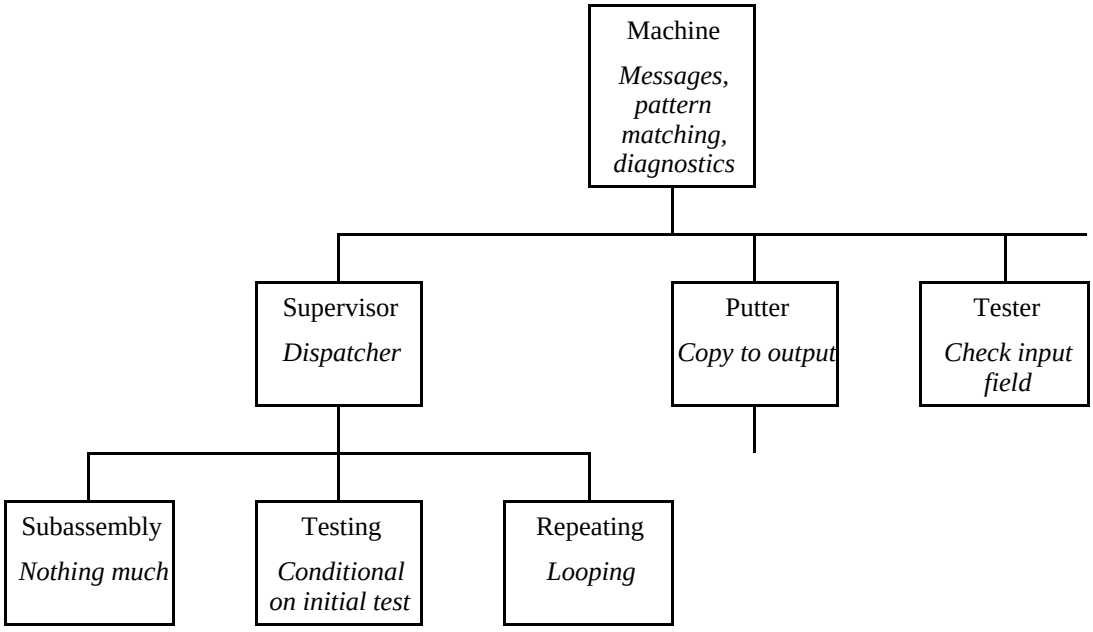
And that is, approximately, that. There is plenty more to say about the system, but unless there are unpleasant surprises in store I've told you how it works. I emphasise once again that it is not optimal, and not even intended to be; it is intended to work, and to give me experience in building this sort of machine. All being well, other similar machines will follow, for there's a lot of documentation management which broadly fits into the pattern explored here. I would like to think that in this exercise I've identified some of the primitive operators which will be useful, and found a way to provide them in useful form which can be built up into larger systems capable of useful work, and expanded if necessary to include further useful operations. Time will tell.

STRUCTURES AND IMPLEMENTATION.

My description so far has not dwelt on structural aspects of the system. There has been mention of some simple structure in the Courses file, and slightly structured diagrams of production lines have appeared, with much the same structure reflected in code fragments here and there.

It is a curious fact that none of this is directly reflected in the real code, and that structures likely to be useful in implementing the system are of a quite different sort. That is because a PDL system is organised as a finite state machine, with one state for each programme instruction, and the sequence of states determined by the machine setting the next state when it has finished the task associated with the current state. The reasons for choosing this organisation have been expounded in some detail[3]. The result is an architecture in which on entry to a state a dispatcher schedules for execution the code associated with that state; before leaving, the code specifies the next state, then returns to the dispatcher. This is not a very structured way of doing things.

There is more structure in the machines themselves, as I have suggested by identifying machine classes (putters, testers, and so on). A hierarchy is readily discernible, of which a fragment of a first sketch might look something like this :



The hierarchy isn't a surprise either; it was suggested from the beginning[12], it was identified as the Machine Classification database, though not otherwise defined, in the report[3], and Adrian went into it in some detail in his thesis[9]. It is gratifying that in this example there is a clearly defined set of functions (some examples are in italic type in the diagram) which can be added at each stage in the hierarchy as the machines become more specifically defined.

The communications facilities required are not demanding; provided that any machine can send a text message to any other machine, the system should be satisfied. Looking (a long way) to the future, it will probably be an advantage if the machines can be widely distributed, but that isn't essential at the moment.

For this implementation, a purely sequential mode of operation is sufficient, but I have observed that there are opportunities for parallel running. In the general case, it is certainly desirable that concurrent execution be possible.

Taking all these factors into account, it seems that a suitable implementation language would be Java, if it worked, which is less than obvious[13]. But we'll see.

AND, FINALLY – THE WORD MACHINE.

The Word machine is another kettle of fish entirely, and I shall not enlarge on it here. For the moment, all it has to do is to report that the assembly process is complete, and the rest can be carried out by hand. I shall describe it later, but this is quite enough for one note. Be patient.

REFERENCES.

1 : G.A. Creak : *The U-BIX operating system*, unpublished Working Note AC44 (October, 1986).

2 : G.A. Creak : *Information structures*, unpublished Working Note AC111 (May, 1997).

3 : G.A. Creak : *Information structures in manufacturing processes*, Technical report #52 (Auckland University Computer Science Department, February, 1991).

4 : N.R. Spooner : *Design and implementation of a Process Description Language (PDL) interpreter* (Auckland University Computer Science Department, 07.485 Project Report, 1995).

5 : Hewlett and Packard in a garage : http://www.hp.com:80/abouthp/HPGarage.html.

6 : G.A. Creak : *Background for a document generator*, unpublished Working Note AC115 (September, 1997).

7 : G.A. Creak : *CROAK*, Technical report #18 (Auckland University Computer Centre, 1980).

8 : R. Kerr : *Knowledge-based manufacturing management* (Addison-Wesley, 1990).

9 : A.S. Krzyzewski : *Knowledge-based solutions in automated manufacturing* (M.Sc. Thesis, Auckland University Computer Science Department, 1992).

10 : G.A. Creak : *States in PDL revisited*, unpublished Working Note AC116 (October, 1997).

11 : G.A. Creak : *Clarification*, unpublished Working Note AC82 (January, 1992).

12 : M. Brodsky, G.A. Creak : "The design of an expert system environment for factory control" (*Proceedings of the Fourth New Zealand Expert Systems Conference*, Massey University, 1990), pages 31-47.

13 : Almost everybody : *Anecdotal communications*, frequently.