

Systematics—a non-programming language for designing and specifying commercial systems for computers

By C. B. B. Grindley*

The paper describes a new language which provides tools and techniques for the systems analyst. The main feature of the language is that, like mathematics, it is an aid to problem solving, but, again like mathematics, it is not provided with a compiler. A short example is given showing how the analyst is enabled to construct a model of the information system he is designing independently of any computer considerations.

In the early days of commercial data processing by computer, there were no systems analysts as we know them today. The job of getting work on to the computer was seen to be only one of programming. True, before he could write and test his programs, the programmer had to get the facts from the user. In the event, this task turned out to be one of major difficulty. How often did the programmer complain that the user did not know what he wanted? Today it is not too difficult to see why this was so. No one person knew all the facts. The clerks did not know how their separate tasks fitted into the broad system. The managers did not know the detailed work that went on within these separate tasks. More important, many of the rules that the programmers were seeking did not exist. Managers, supervisors and clerks frequently exercised their discretion when deciding on a course of action. Instead of saying what rules they followed in each case, they said it was a matter of experience and judgement.

To cap it all, after a while it began to be seen generally that simply to repeat the existing system was not good use of a computer. The old system was designed to suit the capacity of human beings to do work. The capacity of a computer to do work is in many ways greater than human beings', in some ways less. But whether greater or less, it is essentially different. On the face of it, it is unlikely that the old system will do. A new system should be designed to exploit the different capacity of the computer. Gradually the analysis of users' requirements, and the design and specification of an information system to meet these requirements, emerged as a prior and separate task from programming.

Systems analysis

Although the systems analyst now claims recognition as an expert in his own right, his title has not been clearly stated. The overall job of getting work on to a computer is still something of a relay race, where the analyst passes the baton to the programmer just about when he feels he's had enough. In some cases the analyst prepares flow charts which consider quite detailed computer problems, and from which the programmer codes his program. In other cases he simply identifies broadly the output required without

attempting to provide all the detailed rules showing how that output is derived. In a series of studies that Urwick Diebold undertook about two years ago it was found that programmers spent less than half their time actually writing and testing programs. The rest of their time was taken up mainly with delays caused by something called "systems queries". On further analysis, systems queries were found to be due to:

1. Changes made by the analyst to the system as originally specified.
2. Misunderstandings between the analyst and the programmer over the specification.
3. Errors in the specification.
4. Omissions in the specification.

Two things emerged quite clearly. The first was that within the overall job of transferring work on to the machine there are two distinct jobs involved. These jobs are distinct since they serve different needs. Job I considers the user's needs and is concerned with designing an information system. Job II considers the computer's needs and is concerned with designing a programming system which will satisfy these information requirements using a computer. The second thing was that Job II is not just a detailed extension of Job I. Both jobs explore, *in final detail*, the separate problems with which they are concerned. Before the programming function begins to plan *how* a particular job is to be performed efficiently on a certain computer, it must be furnished with a specification showing, for all cases that can arise, *what* is required.†

Designing a system

If the systems man is to provide rules for every case that can arise then he is, in fact, constructing a model of the information system he is designing. Even if such a model were not required before programming could properly be undertaken, it would still be valuable to construct it, because it is only by constructing such a model that the designer is forced to face all the implications of his system. It is a way of trying out his ideas.

† A definition of the separate tasks of the systems analyst and of the programmer is offered in Appendix 1 to this paper.

* Urwick Diebold Limited, St. Andrew's House, 40 Broadway, London, S.W.1.

What are the requirements for model building? They are twofold:

Firstly: to break down the problem into its component parts.

Secondly: to describe precisely the relationship of all the parts with one another.

The scientist frequently has to construct models. He has developed techniques and languages for this purpose, for example, mathematics. The need of the systems analyst is clear. To perform Job I he needs the techniques and languages to construct models of information systems.

Commercial programming languages do not at present meet this need. Let us examine why this is so. It is interesting to compare them with scientific programming languages, for example, ALGOL, which do appear to satisfy the needs of the scientist. This is because his needs were different. True, he needed to construct models. But long before the advent of the computer, mathematics was already developed to the point where it was rich in concepts and techniques for this purpose. Scientific programming languages have been able to make use of existing mathematical notation. The systems analyst had no such language for designing information systems. English existed, but to the extent that programming languages have imitated English they have missed the point. English is entirely unsuitable for model building. It is imprecise and open to interpretation. To illustrate this it is only necessary to look at the version of English used in statutes in an attempt to cover a situation completely, and then to see the lengths to which courts have to go in practice in order to interpret these attempts. The commercial programming language should ideally have provided for the analyst his equivalent of mathematics. But, no matter how great is the claim of any commercial programming languages to be problem orientated, they are, without exception, designed to suit the way a computer goes about processing data. There is good reason for this. It is the aim of such languages to be automatically translatable into a machine's language. Commercial data processing has had to make efficient use of computers. The difficulties of producing compilers to translate efficiently into a machine's language have acted as a restraint upon the free development of concepts useful to the analyst.

Systematics

A new language has been developed to meet this need. It is called *systematics*. This language is solely concerned with techniques and concepts useful to systems analysts in designing information models to meet user's requirements. This has been achieved largely because no attempt has been made to provide it with a compiler. It is thus completely computer-independent. It is interesting to compare this feature with one of ALGOL's most important uses. The 1958 Zurich Conference set for ALGOL three objectives. One was that it could be

used to describe computational processes in publications. This objective has been successfully achieved to the extent that the language is now frequently used to describe processes which it is never intended to perform on a computer. In such descriptions ALGOL expressions are used for which no compiler exists. Essentially, systematics is a tool for specifying solutions to information systems problems. More important, it is also a tool for developing such solutions. Like most tools for scientific analysis, it points to deficiencies in proposed solutions. It shows the analyst where more information is required, where certain circumstances have not been covered, where rules suggested are inconsistent. The models constructed may be of large total systems or of detailed parts of such systems. The system may eventually be performed on a computer, or partly on a computer and partly using other processing methods, or not on a computer at all.

How does systematics work? Whilst not attempting a full description of the language in this paper, the three most important features are described. In addition, a detailed example of systematics applied to a simplified payroll is given in Appendix 2. Before discussing these three features, it should be said that the overall objective of each of them is to give precision to the building of information systems' models comparable to that supplied by mathematics. Why then not use mathematics? In fact, mathematical concepts and notation are used freely where appropriate. Indeed, any existing, precise and standard notation may be used within systematics if found to be useful in the context of the information system being designed, e.g. statistical and Boolean notation. Systematics provides a framework within which these existing concepts, together with a small but growing number of concepts peculiar to systematics itself, may be used. But this framework is designed to suit the problems encountered in information systems design. These problems are different from mathematical problems. They differ principally in two ways. Firstly, in information systems, the relationship between a given result and the information from which it is derived is usually relatively simple. The complexity of the problem arises from the very high number of alternative relationships which may apply depending on the values and states of other items of information. In mathematics the relationship between results and the information from which they are derived is generally far more complex, but the number of alternative relationships is relatively small. Secondly, relationships in information systems depend not only upon quantities, but upon other qualities which we are not accustomed to measure and to which the assignment of numerical values appears inappropriate, e.g. sex, behaviour, location, etc.

Three principal features of systematics

Alternative conditions

The first feature is concerned with the problem of fully exploring all of the high number of alternative conditions

which can arise in information systems. The language thus recognizes two entirely different sorts of statements.

- (i) Statements of condition—Connectives.
- (ii) Statements of derivation—Expressions.

The statement of condition is contained within a Boolean AND/OR matrix. It treats all combinations of the possible states of relevant items of information and shows what action follows each of them.

e.g.	I_1	OR			
		p	q		
	I_2	\vee	x	\vee	x
	AND	E_1	E_2	E_3	E_4

Essentially it says if item 1 is state p AND item 2 is state \vee do expression 1, OR if item 1 is state p AND item 2 is state x do expression 2, OR etc. The statement of derivation or expression simply shows how a derived piece of information is related to the pieces of information from which it is derived, e.g.

Derivative = state of item 1 + state of item 2.

Definition of qualities

The second feature is concerned with giving precision to qualities other than quantity. It provides for two things. Firstly, for each item of information, the way in which its particular state can vary significantly from case to case is shown. Secondly, a name or reference for each of these states is given. For example:

Item	Variability
Area.	Europe: Africa: Rest.

Variability should not be confused with "range", frequently given for each item of information by systems analysts. Range is for checking the validity of each item as read by the computer. Variability is for the purpose of exploring all significantly different states of the items within the connectives just described. For example, if "area" were one of the items in the Boolean matrix, then what to do in the case where its state was "Africa", or "Europe", or "Rest" would have to be shown. And the names Africa, Europe and Rest would have to be used. The quality of area may thus be talked about with precision. It is defined as being one of these three states.

Classification of information

The third main feature is the classification of each item of information according to the part it plays within the model. One method of classification is according to permanence. Four conditions are recognized. Does the state of the item remain unchanged during the operation of the model or does it change. If it changes then is it up-dated, originated or destroyed. Another method is by generation. For those items that change

their state it is shown whether they have yet changed and if so, how many times. A further major classification of the items of information is into a hierarchy of classes, sub-classes, sub-sub-classes, etc. It follows here principles well established in logic, scientific method and in language construction.

The example given in Appendix 2, together with the notes at the foot of the Appendix, should illustrate in more detail the principles of systematics. The language is no more than a specialized branch of mathematics. It provides additional tools which are particularly valuable to the design of information systems. It lays the foundation for a body of knowledge appropriate to this new field of activity. The word "new" is used advisedly since it is only the recent use of the computer for processing information that has made it necessary to design and construct information systems with such precision. Until the appropriate body of knowledge and techniques are reasonably well formulated it is not intended to inhibit the development of systematics by providing it with a compiler.

Appendix 1

Definition of systems analysis and of programming

Systems Analysis (ignoring work other than that to be performed by computer)—Expressing the relationship between the data fed into a computer system and the information to be produced by it.

Programming—Expressing the relationship between the data fed into a computer system and the information to be produced by it in a manner which is efficient in terms of the capabilities of that system and in a manner which can be interpreted by that system.

Implications of definitions

1. The *analyst* requires no computer knowledge. True, before constructing a detailed model of an information system, satisfaction that it is feasible to perform the work on the computer should be obtained as far as is possible. Also when the analyst is concerned with the interface between user and machine, i.e. design of input and output documents, computer considerations are involved. Both these activities will require liaison with the programming function. But the statement remains broadly true. The analyst is concerned with identifying the user's needs and constructing an information system to meet them.

2. The *programmer* is solely concerned with efficiency problems involved in processing the information system on a given computer, and with translating his solution of these problems into the machine's language. Translation from systematics could, of course, be done automatically. The programmer's basic job therefore is concerned with computer efficiency. If, and only if, computers are made where processing efficiency no longer matters, or efficiency problems can be solved automatically, his job will disappear.

Appendix 2

Example of systematics applied to a simplified payroll

LEVEL 1.

THE MODEL

E¹ Calculate Pay

Dictionary

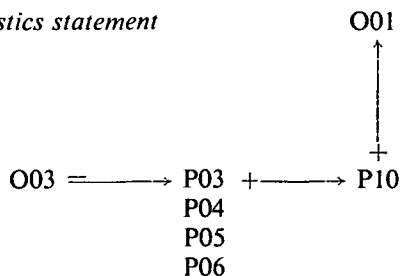
Reference	Description	O	I	R	Variability
E ¹ P01 E	number	✓	✓	✓	NNNN
P02	name	✓		✓	A→
P03	sex			✓	M: F
P04	birth			✓	NN/NN/NN
P05	reduced N.I.			✓	✓: x
P06	pension scheme			✓	O.K.: No N.I.: No G.P.
P07 a	department	✓	✓		NN
P08	marital status			✓	Single: Married: Widow
P09	rate of pay			✓	0d.—25/0d.
P10	date		✓		NN/NN/NN
P11	holiday		✓		0—15
O01	gross pay	✓			N→
O02	net pay	✓			N→
P12 (a)	hours worked		✓		0—80 ($\frac{1}{4}$)
P13	sports			✓	6d.: 1/-
P14	hospital			✓	0d.: 3d.
O03	national insurance	✓			N→
U01 (a)	hours to date			✓	N($\frac{1}{4}$)
U02	gross pay to date		✓	✓	N→
O04	tax	✓			N→
U03	tax to date	✓		✓	N→
P15	tax code			✓	0—999
O05	other deductions	✓			N→
O06	maximum hours	✓			N→
etc.					

LEVEL 2.

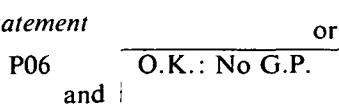
ROUTINES

(only National Insurance shown)

Characteristics statement



Conditional statement



Connective statement

Intermediates

A—Age

B—40 hr. earnings

C—Flat rate

D—Graduated pension

Connectives

Flowchart

A, B

a, C

b

D

O03

a: C =→ A, B, P03, P05.

b: P06, O01.

LEVEL 3.

EXPRESSIONS

A =→ P10 — P04

↓

B =→

↓

a

↓

a

↓

D =→

↓

O03 =→

↓

O03 =→

↓

O03 =→

↓

O03 =→

↓

O03 =→

↓

O03 =→

↓

O03 =→

↓

O03 =→

↓

O03 =→

↓

O03 =→

↓

Level 3, contd.

CONNECTIVES

a: A

P03

P05

B

and

C:

↓

O03

↓

O03

↓

O03

↓

O03

↓

O03

↓

b: P06

O01

and

D

↓

O03

↓

O03

↓

- N.B. (i) Assumed for simplicity that hours worked not zero.
 (ii) Intermediates not calculated assumed to be zero.

Key to symbols used:

\Rightarrow	derived from
$+\rightarrow$	together with
\longrightarrow	followed by
n	not applicable
$ $	the remainder of the range within straight brackets
x	round y and above up to nearest
ry	x .

Notes: Systematics describes the information model at three levels of detail.

Level 1. The model itself

A model is restricted to a major and separate operation performed upon one class only. At this level a broad statement of the purpose of the model is given identifying the information class concerned. This is followed by the dictionary. The dictionary lists and classifies every

item of information used in the model and defines its variability. Permanence is indicated by P = permanent, U = up-dated, O = originated. Subscripted "a" indicates a sub-class. On its own it indicates the principal of that sub-class, i.e. the item of information that uniquely identifies a particular member. In brackets it indicates a subordinate of that sub-class. O, I and R show whether the item is output, input or held as a record, or what combination of these apply.

Level 2. Routines

A separate routine is provided for each up-dated item and for each originated item. It states which other items are required to derive the item concerned and under what conditions the routine is performed.

It also states the "intermediate" steps taken on the way to the final derivation, and how these are related to each other through the connectives. References are provided for each expression (capital letters) and each connective (small letters).

Level 3. Expressions and connectives

The derivation for each routine is stated in final detail at this level.

Correspondence

To the Editor,
 The Computer Journal.

Sir,

The asides, as it were, in J. P. Penny's analysis of a time-shared computer system (this *Journal*, Vol. 9, No. 1, p. 53) raise issues that are by no means settled regarding the preferred or even useful storage organization of such systems.

Towards the end of Section 5 of the paper we have the assertion that the storage requirements of a program can be found exactly when it is compiled. This is certainly not true of programs written in ALGOL, CPL, PL/I, or similar languages, the similarity being that they allow dynamic storage allocation. That this is not an exotic frill only of interest to the far out fringe I can testify after nine years' connexion with commercial and industrial applications of computers. There, there are very many practical jobs that could be greatly simplified, completed more speedily, and made cheaper to run, given full dynamic storage allocation.

In Section 2 we have the argument that the simplicity and inherent efficiency of single level storage is attractive for time-shared systems. But this assumes that it is convenient, let alone economic, to have all of a program, and all of its data space, available in primary storage at the same time. Firstly, most programs above the level of student exercises may have 50 or more of their instructions written only to deal with exceptional circumstances or errors. On any particular run of the program one hopes that most of this necessary lumber will not be called upon. It is therefore not economic to use

single level storage if much of it is not called into use. A similar situation arises with, for instance, periodic analyses appended to a regularly run program. In an ideal world we would have two debugged versions of the program, with and without analysis segments, to run as required. But in practice there are too few programmers to do the tidying up, and multiplying programs harms the administration of the computing facility.

Secondly, most programs have several distinct phases, initialization, close up, main processing cycle, and so on. Although they may share common subroutines only one phase need be available for execution at a time. Thus the logical structure in the way we happen to write programs would seem to be amenable to the reasonably efficient use of multi-level storage systems, as long as the source language programmer is only faced with a virtual single level store of suitably vast size.

Reverting now to time-sharing, it would seem that we can only make efficient use of the large high speed primary storage required for this if we also incorporate in our system very much larger secondary storage to buffer the fluctuating storage requirements of the programs concurrently in the system.

Yours faithfully,

H. D. BAECKER

Imperial College,
 London, S.W.7.
 12 May 1966