THE NEED FOR IMPRECISION

The growing interest in decision tables as a tool for programming business 
dp applications is a symptom of disenchantment with the conventional tech-
niques—by which I mean logic flowcharts and procedural languages. I'm 
all on the side of the disenchanted; these conventional techniques are hardly 
ever the right ones to use in business 
dp; mostly they're used because no alter-
native is available, and because pro-
grammers are pretty conservative peo-
ple anyway. But it seems to me that we 
are overlooking one of the most impor-
tant disadvantages of procedural 
programming, and that the benefits we 
could gain by eliminating it are in dan-
ger of being missed.

The usual indictment runs like this. Writing procedural programs is dif-
icult, and unreasonably so. Given a spec-
fication, the programmer is required to 
devise a logical structure which is 
related to it in a particular, non-ob-
vious way; the complexity of this rela-
tion accounts for most of the difficul-
ties. Devising the structure is itself an 
arduous activity for all but the simplest 
specifications; it is easy to make mis-
takes and, in repairing them, to cause 
fresh errors. The procedure defined by 
the logical structure will usually have 
several properties which the program-
ner did not intend; when conditions 
arise which he has not foreseen, these 
additional properties are likely to cause 
trouble. Also, it is impossible to check 
the operation of the procedure on a 
particular set of input data except by 
simulating the program (or actually run-
ning it); there is no analytical method of 
debugging.

So far, so good; the case for finding a 
better, nonprocedural technique is 
well-founded. But the indictment does 
not go far enough: the worst crime is 
committed in our attempts to mitigate 
the difficulties above. Because pro-
gramming is so difficult, we have been 
driven to adopt a rigorous method-
ology. We decree that specifications 
should be complete and precise; that 
they should be "frozen" while the pro-
gram is being developed; they should 
be self-consistent and exhaustive, and 
without redundancies. We make these 
rules because without them we may not 
be able to write programs at all. But 
they are inherently bad rules, imposing 
constraints that are irksome and 
often unacceptable. As soon as pos-
sible, we should kick over the traces and 
be free.

Unfortunately the prisoner doesn't al-
ways notice when the door of his cell 
is opened; and too often he prefers 
the security of his prison to the more 
demanding air of freedom. In a typical 
article on decision table techniques we 
read: "Tables force the analyst to make 
a complete and accurate statement of 
the problem logic . . . tables provide 
for better optimization, since computer 
programs can check tables for com-
pleteness, redundancy and contradic-
tions." This seems to me to be very 
wrong-headed. We should be positively 
looking for and developing program-
mng methods that do allow inconsis-
tency, redundancy, ambiguity and in-
completeness; we should recognize that 
these seem to be vices only because the 
error-prone techniques of proced-
ural programming make them so.

But allowing that we need no longer 
call them vices doesn't in itself make 
them virtues. It would be ludicrous to 
complicate our dp systems by wantonly 
introducing confusion and inconsistency 
into situations where none existed be-
fore; and we must recognize that some 
problems can only be solved by ex-
treme rigor and precision. My theme is 
concerned with those situations where 
confusion and inconsistency are inher-
ent elements of the problem and where 
we cannot hope to write successful pro-
grams unless we are able to deal prop-
erly with these factors.

Consider, first, those cases where no 
specification can be agreed for the pro-
gram to be written; the most obvious 
instance is machine translation of nat-
ural human languages. We know what 
we are trying to achieve, but we can-
ot pin it down in any but the broadest 
of specifications; arguably, many of the 
most sophisticated attempts to devise 
systems for machine translation have 
failed precisely because they have re-
died on a detailed specification (usually 
of some lexical or parsing algorithm). 
When the specification proves faulty, 
the techniques used allow no substanci-
tial modification without complete re-
design. Programming in this fashion is 
like playing golf with crazy rules—rules 
which demand that if you don't hole 
in one you must go back and drive 
from the tee again; to play like this is 
to miss the crucial point that makes golf 
possible: you get to the hole by a con-
vergent series of strokes, and it doesn't
the forum

matter if you can’t see the hole from the tee.

Consider, next, the specification that is incomplete. In a complex payroll application, for example, the rules determining what each employee is to be paid will be based on legal requirements, on piecemeal agreements with several labor unions, on practical difficulties, such as widely dispersed paying points, and so on. When the systems analyst tries to formulate the programming specifications he discovers that these rules are not easily reduced to an ordered scheme; in particular, his attempts to do so may reveal areas in which the rules are simply not defined at all. He may ask “how is gross pay calculated for an employee on code 17 who is working on a scheduled rest-day when that day happens also to be a public holiday, and the total number of hours worked is less than a normal working day?” And there may be no answer to this question because the case has never been considered before. The analyst has to put the question only because he needs an artificially complete and tidy specification.

Then consider the inconsistent specification. It is common for the rules of a manual data processing system to develop by allowing exceptions to the general rules, then exceptions to the exceptions, and so on. The systems analyst cannot represent this situation correctly by distilling out of it a firm and consistent specification; he needs to be able to describe the system naturally, in its own terms.

Too often in the past computer systems have been designed in defiance of their users’ needs and wishes. It is too easy to castigate the user who isn’t sure what he wants, who can’t define his needs precisely, who seems to be pursuing incompatible objectives. Of course he is often just being muddle-headed about a simple problem, or too lazy to think it out properly; of course he is often pursuing a confused policy that badly needs to be rationalized. But often he is recognizing that the complexity of his task needs a more subtle and flexible treatment than the analyst and programmer seem able to provide. One of our most important aims is moving away from procedural techniques should be to equip ourselves to meet this need.

—M. A. JACKSON

John Hoskyns & Co., Ltd.