THE EFFECTIVE ARRANGEMENT OF LOGICAL SYSTEMS

Edsger W.Dijkstra
Burroughs
Plataanstraat 5
NL-4565 NUENEN
The Netherlands

We all know that when we have to design something "large" or difficult", we have to apply in one way or another the old adagium "Divide and Rule". Our machines are made from components, our programs are made from modules and they should fit together via interfaces. That is fine, but it raises, of course, the questions how to choose the modules and how to formulate the interfaces. This paper explores the main goals of modularization; being aware of them should assist us in evaluating the quality of proposed modularization.

* * *

An inspiring example of modularization outside our own field is the way in which human knowledge is divided over the different scientific disciplines. Why do we teach different disciplines at our Universities? Why don't we teach all our students just "knowledge"? The answer is simple: our human skulls are too small and our days are too short. The amount of knowledge needed for each discipline must fit into a human head. Besides knowledge there are abilities, and human abilities have two characteristics: they take a lot of training before they are mastered, and thereafter the maintenance of their mastery requires that they are nearly daily exercised, for without such daily exercise they fade away. (This, by the way, is one of the explanations why the capable are always so busy.) In this sense, rather quantitative human characteristics impose a set of equally quantitative limitations on what we are willing to consider as a single scientific discipline.

But there are also internal, more structural constraints. I mean that just an arbitrary collection of scraps of knowledge of the right total amount does not constitute a scientific discipline! It must be sufficiently coherent and self-supporting: it must be possible to study the subject matter of a scientific discipline in isolation (or nearly so), largely independent of what is happening or known in other scientific fields. And the increased insight should enhance our abilities, our en-

hanced abilities should assist us in improving our insight.

The above very rough sketch of how mankind as a whole has parcelled out its knowledge has been included because it also provides a model of how, on a microscopic scale, a single scientist works when he focusses his attention on an aspect of his problem. For every problem too large to be solved at a single stroke of the pen we try to apply a similar technique. We try to isolate various aspects of the problem and to deal with them in turn by "concentrating our attention" on them. (The latter does not mean that we study them in complete isolation: through the corners of our eyes we usually still look at all we are temporarily ignoring!)

The usual catchphrase for this technique is "separation of concerns". Although very adequate from a descriptive point of view, it raises of course the same sort of questions as we raised initially about modules and interfaces, such as "Which concerns should be separated?" and perhaps "After separation, how do they combine again?". This similarity is a rather clear hint that the successful "modularization" of a large information processing system is <u>not</u> a trivial matter.

* * *

The discovery that from a "larger" concern, a few "smaller" concerns can be successfully extracted usually ranks as a scientific discovery. Let me mention a few of them from our own field, so that we know, what we are talking about.

a) The isolation of the definition of the syntax in the task of defining programming languages. (John Backus, 1959; as BNF immediately used in the definition of ALGOL 60.)

b) The isolation of logical aspects of operating systems via the model of cooperating sequential processes. (Edsger W.Dijkstra, 1961; quickly thereafter used in the design of the THE Multiprogramming System.)

c) The isolation of programming language semantics computational histories. (C.A. R.Hoare, 1968; immediately used in the axiomatic definition of semantics.)

I think that the above three examples are fairly typical: all three "separations" (or "isolations" or "extractions") have been highly rewarding. Firstly it was quite clear that the people responsible were not just playing a game, they extracted what seemed a very relevant and possibly manageable aspect from a large and burning problem. Secondly they created a real of thought rich enough to have many thoughts in!

Example (a) opened the way for parsing theory, example (b) for the theory of synchronization, deadlock prevention etc., and example (c) has opened the way for practicable techniques for proving the correctness of programs. In all three cases the problems addressed can now be dealt with quite professionally. All three are easily "rich" enough to be the subject of a one-semester course in a University curriculum, and all three are so well separated from other concerns that such a one-semester course could be fairly self-contained.

Yet another observation should be made. By ignoring, abstracting, generalizing (or whatever verb you wish to use to indicate the not taking into account of some facts) a dual goal has been achieved: thanks to it the theories are of a wide applicability and at the same time of an internal simplicity. (Think of the tremendous simplification of the theory of cooperating sequential processes that was made possible by not dragging speed ratios into the picture! If knowledge about speed ratios had been essential, the correctness arguments would have been an awful mixture of discrete and continuous arguments, and it would all have become very complicated.) It has been said that "everything can be regarded as a special instance of something more general", but I would like to add that there is only a point in doing so, provided that the general view simplifies our arguments. This condition is certainly not met when the more general something can only be understood via a case analysis ranging over the different special instances.

* * *

Another inspiring example is provided by the arrangement of mathematical arguments.

Of our various thinking activities I shall reserve the term "reasoning" for all manipulations that are formalized --or could readily be so-- by techniques such as arithmetic, formula manipulation or symbolic logic. These techniques have a few common characteristics.

First of all, their application is straightforward in the sense that as soon as it has been decided in sufficient detail, what has to be achieved by them, there is no question anymore how to achieve it. And whenever such a technique has been applied, the question whether this has been done correctly is undebatable.

Secondly --and this is not independent of the first characteristic-- we know how to teach these techniques: arithmetic is taught at the primary school, formula manipulation at the secondary school, and symbolic logic at the university.

Thirdly, we are very good at doing modest amounts of reasoning. When large amounts of it are needed, however, we are powerless without mechanical aids. To multiply two two-digit numbers is something we all can do; for the multiplication of two five-digit numbers, most of us would prefer the assistance of pencil and paper; the multiplication of two hundred-digit numbers is a task that, even with the aid of pencil and paper, most of us would not care to undertake.

In order to reach a conclusion the amount of reasoning needed is often the stumbling block, and I have yielded to the temptation to relate the effectiveness with which we have arranged our thoughts to the degree in which we have reduced the amount of reasoning needed. A recent experience has confirmed that this seems sensible. I was compiling a collection of what I thought to be impressively elegant solutions. The first thing that struck me was the surprising degree of consensus among my mathematical colleagues: when I asked them for suggestions they often came with the same examples. The second thing that struck me was that, when I showed any of them a solution from the collection that happened to be new for him, I evoked uniformly the same reaction: laughter! The thrid thing that struck me, however, is in this context the most important one: all the impressively elegant solutions were very short. I therefore ask you to subscribe --at least until the discussion after this talk-- my thesis that the effectiveness with which we think is closely related to the reduction of the amount of reasoning needed, because, as soon as you have subscribed that thesis, you will agree with me that it is a good thing to know by what methods we can reduce that amount and by what methods we can increase it: those of the former category are the ones to be applied, those of the latter category are the ones to be avoided.

* * *

An obvious method is avoiding repetition. When multiplying two ten-digit numbers with pencil and paper we constantly appeal to the 10 by 10 multiplication table of the products of one-digit factors. Whether or not we know the multiplication table by heart or have it written out in front of us for quick reference is unimportant for the purpose of this discussion. What is important is that while multiplying those two ten-digit numbers, we have 100 theorems at our disposal, of which $7 * 8 = 56$ is an instance. If we know how to count, or know how to add, we can prove that the product $7 * 8$ equals 56, but that proof requires a certain amount of reasoning, so much as a matter of fact that we would not like to do it over and over again, everytime we need the product $7 * 8$. Hence the knowledge that that product equals 56 is cast into a theorem; together with the other 99 theorems it forms what is known as the multiplication table.

Another remark of a directly quantitative nature is that we would not expect much use for a theorem whose statement is longer than its proof: instead of appealing to the theorem it would be simpler --at least "shorter"-- to mention directly its proof.

The quantitative remarks in the two previous paragraphs, although of some relevance, do, however, not tell the complete story: if they did, there would be no point in stating and proving a lemma that is used only once, and there is a point in doing so.

Suppose that the total proof of a Theorem consists of two parts:
A: a proof of the Theorem based on the validity of a Lemma, and
B: a proof of aforementioned Lemma.
If both proofs are correct, the Theorem has been established, but suppose that part B is shown to contain a flaw. If we cannot correct the flaw, or perhaps even discover that the Lemma does not hold, we are in bad shape. If, however, we can correct the flaw in part B , its correction is the only thing that needs to be done: part A survives unchanged and unattended. Thinking about the last scenario I have come to the conclusion that its likelyhood is, all by itself, a sufficient justification for splitting up the total proof --straight from the axioms, so to speak-- in part A relying on a Lemma and a part B establishing that Lemma, even if part A refers only once to it. The conclusion seems to be that we not only seek to reduce the amount of reasoning eventually needed when all would have gone well, but also the amount of reasoning to be expected in view of our fallibility.

But, again, there is more to it. Splitting the total proof into parts A and B , connected by a Lemma used in A and proved in B means
1) that we can study B ignoring A, i.e. ignoring the way in which the Lemma is used: we only need to consider what the Lemma asserts
2) that we can study A ignoring B , i.e. ignoring the way in which the Lemma is proved: again we only need to consider what the Lemma asserts. Because the statement of the Lemma can be expected --see above-- to be shorter than its proof, also here we have to take less into account, and that is nice in view of another human limitation, i.e. the limited span of our attention.

Such a separation of concerns is, however, not only nice in view of our limited span of attention, it has a much profounder consequence. It gives us the freedom of replacing part B by a shorter or nicer proof of the Lemma as soon as we find one, it gives us the freedomg of replacing part A by a nicer or shorter proof of

the Theorem as soon as we find one. As long as the Lemma remains the same, changing one part does not invalidate the other. This observation makes it abundantly clear --at least in my mind-- that we should not regard the appeal to the Lemma as it occurs in part A as an abbreviation of its proof as described in part B . The appeal to a lemma is to what the lemma states, and not to its proof: the main purpose of the introduction of the explicitly stated Lemma was precisely to make part A a consistent whole that is independent of the particular shape of B .

* * *

The above must sound very familiar to every mathematician that has been trained always to try consciously to present his proofs both as concise and as clear as possible. (That not all mathematicians have been trained that way, is another matter that need not concern us now.) Computing scientists --the other designers of what I called in my title: "logical systems"-- seem, amazingly enough, to be in general less aware of it. They are in general aware of the circumstance that what on one level of detail can be regarded as an unanalysed whole, can be regarded at a next level of greater detail as a composite object, they are often not fully aware of its implications. The analogy with mathematical proofs tells us that whenever we regard a whole as composed of parts, the way of composition must define how the relevant properties of the whole depend functionally on the properties of the parts, without the need of taking their internal structure into account.

Let me give you one of my cherished examples. We consider a program part S for the computation of the remainder, more precisely, a program part S satisfying for constant c and d :

$$(c \geq 0 \text{ and } d > 0) \Rightarrow wp(S,\ r = c \bmod d) \qquad (1)$$

(in words: $c \geq 0$ and $d > 0$ implies the weakest pre-condition for the initial state such that activation of S is certain to establish a final state satisfying the post-condition $r = c \bmod d$). Consider for S the following program part:

$$\begin{aligned}
&\{c \geq 0 \text{ and } d > 0\}\ r, dd := c, d; \qquad (2)\\
&\quad \text{do } r > dd \rightarrow dd := 2 * dd \text{ od};\\
&\quad \text{do } dd \neq d \rightarrow dd := dd / 2;\\
&\qquad \text{if } r \geq dd \rightarrow r := r - dd\\
&\qquad \ [] \ r < dd \rightarrow \text{skip}\\
&\qquad \text{fi}\\
&\quad \text{od } \{r = c \bmod d\}
\end{aligned}$$

Many programmers, I have discovered, don't hesitate to consider this program part S as composed (primarily) of three parts, viz. of the form

"S" = "S0; S1; S2"

i.e. the outermost syntactical decomposition. The point, however, is that nowhere the properties of these parts S0, S1, and S2 have been stated, on account of which we can conclude that S satisfies (1). This point becomes a problem as soon as it is discovered that program (2) is wrong. It contains a well-engineered bug: in those cases where c divided by d leaves a remainder = 0 and, in addition, the quotient is a power of 2 , the final state satisfies r = d instead of r = 0 . As it stands we can only conclude that program (2) as a whole is wrong; we cannot --although regarding it as a concatenation of three statements-- decide which of these statements is in error. That question is void.

And, as a matter of fact, we can repair it in different ways. Either we replace S1 by

"do r ≥ dd → dd:= 2 * dd od"

or replace S2 by

"do r ≥ d → dd:= dd / 2;
do r ≥ dd → r:= r - dd od
od"

If we had chosen the properties

$$P0 \Rightarrow wp(S0, P1) \quad , \quad P1 \Rightarrow wp(S1, P2) \quad , \quad \text{and} \quad P2 \Rightarrow wp(S2, P3) \qquad (3)$$

with P0: $c \geq 0$ and $d > 0$

P1: $r \bmod d = c \bmod d$ and $(\exists\, i: i \geq 0: dd = d * 2^i)$ and $0 \leq r$

P2: P1 and $r < dd$

P3: $r = c \bmod d$

then the bug would have been localized in S1 as it may fail to establish P2 .

Note. On account of the semantic definition of the semicolon

$$wp("S1; S2", R) = wp(S1, wp(S2, R))$$

we derive

$$wp("S0; S1; S2", R) = wp(S0, wp(S1, wp(S2, R)))$$

and conclude that (3) indeed allows us to derive

$$P0 \Rightarrow wp("S0; S1; S2", P3)$$

(End of Note.)

The moral of the story is that we can only claim that a whole has been properly composed of parts provided the necessary properties of the parts have been decided. Without that specification it is, as if we are trying to build up a mathematical theory while giving only the proofs of our theorems and lemmata, but not the theorems and lemmata themselves! I have, therefore, decided for myself that such specification of the parts is an absolutely essential constituent of the whole design. After all I have said, this decision may strike you as obvious, as nearly trivial. I am unhappy to report that it is not generally accepted. Quite regularly I see texts arguing that "we cannot impose upon the poor programmer the additional burden of also supplying the assertions that correctness proofs (still) need". Depending on the attitude of the writer, either today's proving techniques are blamed for "still" needing such assertions --in that case the author usually does not mention what alternative proving techniques he has in mind-- or the author now proposes to apply Artificial Intelligence techniques for deriving such assertions mechanically. I hope to have made clear why I regard the latter as a somewhat nonsensical activity; the assertions reflect an explicit choice of the designer, a responsibility that certainly cannot be delegated to a guessing AI-system. (For instance: in the above example we could have replaced in P1 and in P2 , or in P1 only, the term r mod d = c mod d by the more stringent r = c .) An "automatic specification guesser" that is only fed with a single instance of each part is almost certainly bound to be overspecific, and the whole activity strikes me as putting the cart before the horse.

Example. Given the following proof:

"The theorem is obvious, because

$$(x1 - x0)(x2 - x3) + (x2 - x0)(x3 - x1) + (x3 - x0)(x1 - x2) = 0 \quad " \ ,$$

can you guess the theorem? It is --this is a hint-- a very well-known theorem that is usually proved in a rather indirect way. (End of example.)

* * *

I have shown you a small example, specially manufactured to illustrate the nature of the dilemma. Let me now turn to a more grandiose example that has been provided by "the real world". The original design of the IBM650 had the very special feature that the attempted execution of a special little program loop blew one of the fuses of the machine. Needless to say, this very special feature was not mentioned in the manual, but, programmers being as they are, they not only discovered it, they also used it in at least one organization, where reservations of machine time were extended with the down-time, when the machine broke down during your period of reser-

vation. Programmers who had a one-hour reservation for a debugging session used the little loop when, after ten minutes of testing, they discovered a bug whose patching required some peaceful thinking!

The decomposition into the two parts "hardware" and "software" is certainly a time-honoured one, but in this case it was defective. As soon as the aggregate whole was required not to blow fuses while yet it did, none of the two parts could be proved to be wrong or to be correct. The maintenance engineer could argue that all programmers knew that the machine was such that that little loop would blow a fuse and that, therefore, they should not include it in their programs. The programmers, from their side, could argue that the manual nowhere stated that upon the attempted execution of that little loop the machine had to blow one of its fuses! They could throw the blame on the other party indefinitely, and the only way to end this ping-pong game is by choosing a well-defined interface. Either it is decided that the fuse should not be blown --which means a change in the hardware design-- , or it is decided that the fuse should be blown, and then all programmers have the obligation to program around it.

I definitely prefer the first alternative, not so much because I am more of a programmer than of a circuit designer, but because in the interplay between hardware and software we have the greatest variability at the software side. It is therefore simpler to propagate the obligation of fixing the bug through the limited number of machines than through all the programs that are or will be made for that type of machine. I have the sad impression that in user communities, management often takes the undesirable decision and obliges its programmers to program around such deficiencies.

The story about the fuse is old and nearly forgotten. We should, however, remember it as a paradigm for the sad situation in which the majority of today's programmers are supposed to do their work. I know of a respectable colleague who, in the year of the Lord 1976, is developing the basic software for a machine of a less respectable design: its hardware specifications are so obscure that quite regularly he has to get access to the prototype in order to discover experimentally what some commands are supposed to achieve! He has to do so under the assumption that the prototype is in perfect working condition, the trouble, of course, being that, logically speaking, that "perfect working condition" is, as yet, undefined. Together with the hardware designers he has to decide "after the fact", which machine they should have had in mind to build. I like to believe that this is an extreme case, but have no grounds for doing so.....

The complete functional specification of a machine must be given without any reference to its internal structure if, in the whole system, the machine is to be recognized as a part of a composite whole. This, however, does not only apply to hardware --"concrete machines", if you like-- , it is equally applicable to the abstract machines known as higher-level programming languages. Only when their semantics are fully defined without any reference to implementation details such as compilers, binders, interrupts and what have you, only then has the separation of concerns been effectuated that makes any progress possible. It is in this respect disheartening to observe that many a modern high-level language user is much worse off than the average programmer a quarter of a century ago. In the old days programmers used to have a complete functional description of their machine at their disposal and, as a result, they could know exactly what they were doing. This is in sharp contrast to the majority of the so-called high-level programming languages, the semantics of which are only so loosely indicated that most young programmers have lost the sense of complete control that we used to have. They live in a woolly environment in which the notion that a program is either correct or not is, by definition, not applicable.

A politically complicating factor is that the world's largest computer manufacturer has probably not the slightest incentive to change this state of woollyness of the programming languages it supports, because this woollyness only acts at its advantage as long as its products are accepted as what are euphemistically called "de facto standards". In the case of a well-defined programming language, it would have the obligation to implement that correctly and would run the risk of a competitor providing a better implementation; as things are its implementations are taken as "the definition". Such political considerations make its unwillingness to support such well-defined languages as, say, ALGOL 60 only too understandable.

So much for the ill effects of lacking specifications.

* * *

I hope that in the above I have convinced you that, in the invention of the complex composite systems we are considering, a rigorous definition of the essential properties of the parts is not a luxury but a necessity. In the final part of my talk I would like to tackle the more elusive question "By virtue of what type of properties can parts be nice or ugly?" It is the question what interfaces to invent. I called this question "elusive" because it is as impossible to give a final answer to it is impossible to teach young mathematicians how to discover beautiful theorems. What we can do --and, in fact, do while teaching mathematics-- is explaining why we

think that some theorems are beautiful. In a similar vein we should be able to explain to young computer scientists what virtues to look for when they are evaluating or considering a proposed set of interfaces. It is in this connection a good thing to remember that one of the main roles of the decomposition of a whole into parts was the localization of the bug when something went wrong.

Again, let me start with a very simple example. Suppose that we have to control a printing device that accepts 27 different commands --say the printing of the 26 letters of the alphabet and the blank-- . If we control this device with a ternary machine, each command could be specified with three ternary digits because $3^3=27$. But suppose now that we are to control such a device with a binary machine. We would then need five bits for a command. Immediately the question arises what to do with the nonsensical remaining $32 - 27 = 5$ possible "commands". One answer would be that it does not matter because no correct program would ever send any of these five nonsensical commands to the device. But this would be a very silly choice, for it would give the designer of the device the licence to make it in such a way that a nonsensical command could cause a jam in the printing mechanism, and as soon as he has done that it is possible to write erroneous programs that, besides not producing the right results, may wreck the installation. Such a silly interface would cause the ill effects of an erroneous program to spread. Another possibility would be to postulate that such nonsensical commands are ignored by the device. That is safe as far as the working condition of the device is concerned but it is still silly: presumably it was not the programmer's intention to send such silly skip commands to the printing device and, if such a command is sent to the device, we may assume that something in the execution of his program has gone wrong. The sensible reaction is, on the one hand to protect the device from being wrecked and on the other hand to signal an error, thus giving the programmer another way of protecting himself. An alternative way of doing away with the problem would be to extend the character set of the printing device with another five characters.

This, again, was a simple example, specially manufactured to illustrate the problem; but it we look for it, we can find the silly choice made many times, and on a much larger scale. A famous example is the coupling to a general purpose computer of peripherals that may signal to the central processor that they require a certain service from it within a given number of milliseconds; in the worst situation the irrecoverable malfunctioning that results when the central processor fails to meet its real-time obligation is not even signalled! In any case the real-time obligation of the central processor with respect to such a peripheral places a heavy and ugly burden upon the system designer who, for instance, must guarantee that the interrupt

is never disabled for too long a period of time.

We find the same flaw when compilers accept syntactically incorrect programs without warning or when system integrity relies on the correctness of the compilers used.

The quoted examples are instances of a general case. We are dealing with classes of strings: strings of characters representing source program, strings of words representing object programs, strings of commands controlling a device, etc. Either such a class of strings contains all the strings that are physically possible, as in the case of coding the 27 commands to the printer with three ternary digits. In this case there is no redundancy, and we note in passing that under many circumstances such an absence of redundancy is undesirable. Or --and this seems to be the much more common case-- the class of intended strings does not contain all the ones that are physically possible, i.e. our intended strings are represented by the physically ones with a certain redundancy. Using the terms "legal" and "illegal" for strings within and beyond the intended class respectively, we can formulate the following conclusions.

1) The class of legal strings must be defined precisely. If this already presents serious problems, this is a warning not to be ignored.

2) Any part processing such a string should establish whether the string is legal or not. If this presents serious problems, this is a warning not to be ignored.

3) Any part processing such a string should not react upon an illegal string as if it were a legal one.

4) Processing an illegal string may not wreck the part: none of the relations which are carefully kept invariant during the processing of legal strings may be destroyed by the processing of an illegal string.

Note. If program component B processes a string produced by program component A without satisfying the above conditions 2 through 4 --for instance because it is felt to be too expensive to make component B that way-- , we should regard components A and B as belonging to the same part. (End of Note.)

* * *

To wind up I would like to make two suggestions: I would like to suggest to programmers that they have a closer look at mathematics, and to mathematicians that they have a closer look at programming. By virtue of their tradition mathematicians

have a more explicit appreciation for what it means to be rigorous, as a result of the power of currently available machines programmers are more aware of the problems created by sheer size.

I do not suggest that programmers should stuff their heads with mathematical results, but I would like them to get a better appreciation for how and how effectively mathematics are organized. If they do so I expect them to discover that many current topics in computing science are in a sense null-problems as they address problems that should never have been there to start with. I expect them to discover that if they are problems now, we are suffering from the pains of a hell into which our own sins have sent us. I also expect them to discover that these problems are only soluble by starting over again, and that the perpetuation of some old mistakes is the worst mistake we can make now. As very likely candidates for null-problems I mention those associated with compatability, portability and protection.

I also feel that many a mathematician could profit from the exposure to programming. I have that feeling because, while studying mathematical texts, I now very often observe as my reaction towards the author "He must be a very poor programmer!". We, as programmers, have, for instance, been so trained to avoid case-analysis like the plague that it is not unusual at all to encounter a mathematical argument, the length of which can be halved, perhaps even be halved a number of times.

While preparing this invited speech I had to guess what type of audience I would eventually address. The title of the Symposium's subject "Mathematical Foundations of Computing Science" was my only indication. If I have the privilege of addressing a primarily mathematically interested audience, it is clear how my ending note should sound, for in that case I can only urge you, Mathematicians, <u>not</u> to confine with respect to Computing Science your interest to its foundations! The praxis of computer programming needs you as much as you needs it challenge, if Mathematics is to remain the Queen of Sciences.

May 1976
NUENEN, The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow