Software Development as an Engineering Problem

Michael A. Jackson, London

Key-words: Design, development methods, implementation, software engineering, standardisation.

Abstract: It is hoped that software development can become a branch of engineering, but there are important differences. Software is intangible, complex, and capable of being transformed by a computer. Much effort has been devoted to overcoming the difficulties due to intangibility and complexity, but too little has been devoted to exploiting the third characteristic. A process-oriented view of software may lead to substantial improvements in this and other respects.

Stichworte: Entwurf, Entwicklungsmethoden, Implementation, Software Engineering, Standardisierung.

Zusammenfassung: Es wird allgemein erwartet, dass sich die Softwareentwicklung zu einer ingenieurmässigen Disziplin wandelt. Aber hier gibt es noch einige wichtige Probleme: Software ist immateriell and komplex, and sie kann mit Hilfe eines Computers transformiert werden. Es sind grosse Anstrengungen gemacht worden, die Schwierigkeiten, die sich daraus ergeben, dass Software immateriell and komplex ist, zu überwinden, jedoch viel zu wenig, um auch die dritte Eigenschaft auszuschöpfen. Eine prozessorientierte Betrachtung von Software kann in vielerlei Hinsicht zu wesentlichen Verbesserungen führen.

The phrase 'software engineering' was chosen as the title of the NATO conference which took place in Garmisch in October 1968 [1]. The phrase was '... deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering'. Fourteen years after that conference, it is still hard to resist that implication. We still need a wide range of practical disciplines, and practical disciplines need theoretical foundations. We still look wistfully at the established branches of engineering, hoping to model our own activities after the pattern that has served them so well.

But it is not clear that there is any single pattern. There are many branches of engineering, differing greatly one from another. The automobile engineer, designing a new automobile model, is doing something very different from a civil engineer designing a dam; an electronic engineer has little in common with a chemical engineer, an aeronautical engineer with a mining engineer. We should not necessarily expect that a software engineer will be like all of them, when they are so unlike each other. There is no one discipline of physical engineering; instead, there are many disciplines, each broadly characterised by the products which its practitioners are competent to design. Should software engineering model itself on one of these disciplines? If so, which one? Or should software engineering be more eclectic, taking what it can from each of the established branches of engineering to form a new synthesis? Perhaps the idea of a single discipline of software engineering is itself mistaken. Perhaps we need to recognise distinct branches of software engineering, formalising the specialisations that are already apparent: certainly there are specialists in compiler construction, in operating system design, in various aspects of artificial intelligence, in computer graphics, and in several other fields. It may be that no single discipline of software engineering can apply to all of these fields without being so general that it is useless for practical application: theory can be general, but practice must be specific. Even so, at the present early stage in the development of software engineering we can reasonably direct our attention to those characteristics which are common to all or most software, drawing the appropriate contrasts or analogies with branches of physical engineering, and hoping to increase our understanding of present practice and future possibilities. The most obvious characteristic of software is its intangible, immaterial, nature, distinguishing it sharply from the products of physical engineering. A software product is no more a reel of tape or a floppy disk than a piece of music is the paper on which its score is written. From this intangible nature many consequences flow, some beneficial and some potentially harmful. Another, related, characteristic is that the software product is itself capable of being reproduced, analysed, and transformed by computer; it seems that we have not yet taken full advantage of this capability, especially the capability of transformation. A third characteristic is a high level of complexity: most software is very complex, and a central purpose of software development methods is to master this complexity, to allow us to build software that is complex but highly reliable.

These characteristics, and some of their consequences and implications are discussed in the sections that follow. A concluding section draws together the threads of the discussion, and suggests some possible directions for future development.

1 The Intangible Product

The engineer whose product is a physical object works under severe constraints. He must take account of the physical limitations of the materials he uses, choosing appropriate material for each purpose from a relatively small set of available materials. A load-bearing wall must be built of a material with a high compressive strength; the chains or cables of a suspension bridge must have high tensile strength: a window must be transparent; an automobile tyre must be resilient. Both the chosen material and the shape into which it is formed must be adequate to the purpose. The engineer must also consider how the materials will behave over the lifetime of the product. Some materials corrode; some cannot withstand high or low temperatures without long-term degradation; the junction between two dissimilar metals may suffer electrolytic action; a building material may be vulnerable to wind erosion.

The software engineer, whose product is intangible, is almost entirely free from such constraints. The ultimate constituents of the product are the statements of the programming language, and these constituents are, in effect, infinitely strong and entirely free from degradation over time. The addition operator works just as well on large numbers as on small numbers; it does not become worn out after a million additions; it works equally well whether the adjacent operator is a multiplication or an assignment This freedom from constraint is at once an advantage and a difficulty. The advantage is that quite a small set of statements can be used to program any computable function; the difficulty is that there are infinitely many ways of constructing any program, and too few criteria for choosing one way. More constraint is needed.

The need for more constraint has been recognised from the early days of computing. M. V. Wilkes, writing about his experiences [2] in machine-language programming on the EDSAC in 1950, gives a nice illustration:

> "... the integration was terminated when the integrand became negligible. This condition was detected in the auxiliary subroutine and the temptation to use a global jump back to the main program was resisted; instead an orderly return was organised via the integration subroutine. At the time I felt somewhat shy about this feature of the program since I felt that I might be accused of undue purism, but now I can look back on it with pride."

Essentially, the recognition was that a program must be constructed of larger constituent parts than the elementary programming language statements, and that these larger parts must themselves be fitted together in an orderly fashion.

An immediate question arises: what should these larger parts be? The implicit answer for Wilkes in 1950 was that the parts should be subroutines; Wilkes is credited with the invention of the subroutine. This answer was widely accepted for many years, and still has much validity. Fortran, Algol 60, COBOL and PL/I all offer a subroutine construct as the primary part type for program structuring; modular programming and some simpler versions of structured programming rely heavily on it also.

Subroutines, or procedures, can be seen as enlarging the repertoire of machine operations. A machine without a hardware multiplier can be equipped with a multiplication subroutine; a machine without matrix arithmetic can be equipped with a set of subroutines providing the necessary matrix operations; a machine without a 'calendar date' data type can be equipped with subroutines to compare, increment and decrement dates. This is a powerful idea, but much less than a complete solution to the difficulty. With subroutines we can build a sequential process, but there is a need for concurrency also. The essence of concurrency is that during execution of a program two of its parts can both be in non-initial states, although neither is a subroutine of the other. A crude form of concurrency in this sense can be obtained by using 'own variables' in procedures: but this is an uncomfortable technique and leads to obscure and difficult programs. A far better solution is provided by coroutines, which allow a program to be constructed from parts which are themselves sequential processes, communicating by passing control from one to another [3].

The availability of the sequential process as a part type frees the software developer from the unwelcome need to view every problem and every program as a hierarchy. Procedures must be organised into hierarchies, but processes are naturally organised as networks: increasingly, as we tackle more ambitious development tasks, we find problems that can not be readily fitted into the hierarchical mould. Communication among sequential processes need not be by direct control flow, as in coroutines. Processes may communicate by shared variables [14], with suitable provision for synchronisation and mutual exclusion; they may communicate by sending and receiving streams of messages [5]; either buffered or unbuffered; they may communicate by shared events which require the participation of two or more processes [6]. All of these forms of communication allow a 'true concurrency': any process may proceed at any time if it is not held up by communication with another process and if a processor is free.

A different line of development leads to the idea of program parts which are instances of abstract data types [7]. The part is a data object, packaged with the operations which can be performed on that object; the definition of an abstract data type hides information about the representation of the object and the implementation of its operations.

We have here been considering procedures, processes, and abstract data types as general constituent parts from which programs may be constructed; their value, from this point of view, is that they offer the possibility of development in terms of a smaller number of larger parts rather than a larger number of the very small parts which are the statements of the programming language. There is an entirely different question which we will consider later: given a set of part types, and a specific problem, which parts of which types are needed to solve the problem, and how should they be connected together?

2 Designing and Building

In the established branches of engineering there is a clear distinction between designing a product and building it. The design work is an intellectual activity, using intellectual tools and techniques on intellectual material. The building work is a physical activity, creating physical products from physical material. This distinction between the intellectual and the physical has important consequences.

One consequence is that the activities are naturally separated, and performed by different people. The engineer who designs a bridge does not build it with his own hands; he passes his design to a separate construction organisation. The automobile engineer who designs a car does not work in the factory that produces it. Because of this separation of people and responsibilities, and because of the need to deal correctly with the characteristics of the physical materials used, the design must be fully detailed. The automobile engineer does not leave the factory manager the freedom to decide the shape of the engine's combustion chamber or the size of the wheels; the civil engineer does not allow the builder to choose the quality of steel or the formulation of the concrete. The interface between design and building is, within the limits of human fallibility, exact.

Another consequence is that costs can be allocated between the two, and that the costs of physical construction are almost always much greater than the costs of design. Building a bridge costs much more than designing it. Where a product is made in large numbers, as are automobiles, aeroplanes, computers and washing machines, the cost of the complete production run is what matters here, not the marginal cost of one additional unit. Because the physical production cost is so high, the design cost is a relatively small part of the total cost. It is therefore perfectly reasonable to carry out the whole design work more than once before embarking on production. Many automobile designs reach the stage of completed prototypes and are then abandoned without a single production model being built.

A third consequence is that the engineer's work can be examined both in its intellectual and in its physical manifestations. A colleague can look at the design documents critically, checking the design before it is committed to production. Customers and competitors can, and do, look at the finished physical product, discerning its internal structure and design as well as its behaviour in use. Most physical engineering products make their design public property in his sense, and the engineer could not hide the design even if he wished to. The automobile engineer cannot conceal from his customers and competitors the fact that he has decided to place the engine transversely, or that he has used independent rear suspension, or that the engine has six cylinders.

The situation is very different in software engineering. The internal structure of a software product can be largely concealed, especially if the product is delivered in directly executable machine language form. Many of the designer's choices are therefore largely hidden from the world, and, in particular, from his customers and competitors.

The production costs of software are relatively small. Where a software package is produced and sold to many customers, the marginal cost of producing each copy is entirely insignificant. Within the development process itself, we may consider there to be a distinction between designing and building; but however we draw that distinction we will find that the cost of building does not dominate the cost of designing as it does for the established branches of physical engineering.

Nor is it clear that the distinction between designing and building exists at all for software. Both the design and the product itself are intellectual and intangible, and no convincing separation can be made. There is no point at which the development team puts away the drawing board and begins to use the lathe, or to pour concrete. Some efforts have been made to separate design from programming, sometimes by defining a design language which is distinct from the programming language. For example, the designer might produce a hierarchical diagram showing that the program is to consist of certain procedures connected in a certain way; or a network showing processes and their connections [8]. But such a design is grossly incomplete; it is like a design for a bridge which states only that the bridge is to be a suspension bridge with two piers 50 metres high and a span of 250 metres. The separation that has been made is not a separation of design from programming; it is rather a separation of preliminary design from detailed design. Other design languages have been proposed which are essentially forms of 'pseudocode': the design is itself a program, but a program written in a language for which no compiler is available [9]. If the design is complete, it is appropriate to obtain or create a compiler for the design language, thus automating the building activity. If it is incomplete, then again the separation is merely between preliminary and detailed design. In software, the design is the product.

3 Specifying and Implementing

Abandoning the attempt to distinguish design from building, we will use the term ‘implementing’ to replace both of them. It seems fruitful then to draw a different distinction, between specifying and implementing.

A specification states the criteria by which the product will be judged. For example, an auto amplifier may be specified by stating the required gain, harmonic distortion, noise level, output power, and so on. A bridge may be specified by stating traffic patterns, height above the channel which it crosses, the roadways to be connected, and so on. A procedure to compute the *sin* function may be specified by stating the range and format of the argument and the precision required in the result. An inventory control system may be specified by stating the required reduction in inventory holding costs and the required service level for a given pattern of customer demands. It is then the task of the engineer to find and implement a satisfactory solution to the stated problem. But these are all implicit specifications, and are not typical of the specifications found in software engineering, which are most often explicit rather than implicit. C. B. Jones defines [10] the difference:

> “The explicit [specification] is analogous to a program. ... The essence of an implicit specification is to state the relationship required between arguments and results without having to write an explicit rule for computing the latter from the former.”

We could not, for example, give a implicit specification of a payroll program or system; we must specify exactly the rules for computing the gross pay and the tax and other deductions under all the circumstances that can arise. We cannot give an implicit specification for a syntax checker: we must specify all the acceptable syntactic combinations, usually by giving the grammar of the language to be checked.

This need for explicit specifications has caused a difficulty in separating out the specification task from the rest of the development activity, much like the difficulty of separating design from programming. This has been especially evident in data processing, where specifications have often been written which are essentially natural language programs, leaving the customer dissatisfied with a specification he cannot understand, and the design or programmer dissatisfied with the narrowing of his task to that of mere translation. It has seemed that, just as the design is the product, so the specification is the design.

4 Processes and Procedures

But the specification is not the design — or it should not be. We can certainly distinguish between the exact specification of a set of connected sequential processes and the arrangement of those processes so that they can run efficiently on the machine. This will be a natural way of viewing the development of software systems whose subject matter is sequentially ordered in time. Specification captures the time ordering of the real world events; implementation rearranges the ordering, within the freedom given by the specification, to fit the machine. Consider, for example, a payroll system, whose subject matter is the behaviour of the employees to be paid: their working hours, perhaps their production achievements, their holidays and periods of sickness, their promotions, their eventual retirement. It is natural to specify such a system by stating the time-ordering of the events affecting one employee, and the entitlement to pay based on those ordered events. The resulting specification is a sequential process, whose execution models or simulates the behaviour of the employee over the whole period of employment. For an organisation with 10,000 employees, there would be 10,000 instances of this process to be executed. No existing operating system is capable of running these 10,000 processes directly with acceptable efficiency; it becomes the task of the implementer to rearrange these processes into a form allowing efficient execution. Typically, this form will be a set of 10,000 'employee database segments' together with an updating procedure which is executed on a database segment whenever an event occurs for the associated employee in the real world.

We can regard this rearrangement essentially as a scheduling, at implementation time, of the set of processes. The most important aspects of the rearrangement will be:

- conversion of the sequential processes into procedures, so that the behaviour of a process when a relevant event occurs can be treated as an execution of an invoked procedure;
- choosing a representation of the process activation records in terms of a suitable database system;
- choosing a scheduling of the processes so that response to each event is fast enough and the whole system runs efficiently.

Conversion of the processes into procedures can be mechanised; choice of the database representation will depend on the particular database system to be used; the chosen scheduling of the processes must be expressed in an explicit scheduling algorithm to be devised by the implementer.

5 Development Methods

In software engineering we pay much attention to the subject of development methods, certainly much more than is paid in the established branches of engineering. Indeed, it sometimes seems as if there is little in software engineering other than development methods. Where other engineers seem to talk about the products of their activities, and the characteristics of those products, software engineers talk about their activities directly, and about the characteristics of those activities. In part, this is due to the intangible nature of the software product and to the lack of a physical manifestation that can be readily examined and critically evaluated. In part, it is due to the comparative youthfulness of the field, and to the lack of self-confidence in its practitioners: social scientists too spend a lot of time discussing their methods. In part, it is due to the combination in software of great complexity with a stringent requirement for correctness.

Ideally, we would like to have a fully algorithmic method of software development, in which each step is predetermined and each decision can be reliably deduced from what has gone before. This can be achieved for certain problems that are very well understood. The designer of a small transformer, given a statement of the power and of the input and output voltages, can deduce the number and form of the necessary iron laminations for the core and the gauge and number of turns in the primary and secondary windings. The designer of a syntax analysis phase of a compiler can construct the parsing program directly from the specification of the grammar. The designer of a conventional batch program to update a serial master file from a serial transaction file can deduce the algorithm for matching the file records from a specification of the files themselves.

But these are very simple problems. Their solution can be — and has been — automated, and it then ceases to form a significant part of the development activity. Our interest centres on those development decisions which are not apparently amenable to automation, both decisions in the specification and decisions in the implementation activities.

We may regard a development method as a structuring of the development decisions. Here we mean 'decision' in a wide sense, covering the explicit consideration and recording of any relevant fact or choice. It would, in this sense, be a decision in the specification stage of a program to generate prime numbers that the product of two primes cannot itself be a prime; it would be a decision in developing a data processing system for insurance that each policy must be renewed annually; it would equally be a decision that there should be a subsystem for claims and another for premiums, or that a particular subroutine should have certain parameters and should call certain common subroutines, or that the policy master records should be held in an index sequential file.

Different decisions will have different characteristics. Some can be easily taken and carry little risk of error, especially where they record facts which are known *a priori*. Some will be error-prone, but have very limited impact on later decisions. Some, such as a decision to decompose a system into two subsystems, will have a very wide impact. Obviously, there are some general methodological principles which should govern the arrangement of development decisions into a development method. Decisions which record facts known *a priori* should be taken early, before decisions which record choices within the developer's discretion. High risk decisions should be postponed as long as possible, because it will be easier to make them correctly in a later than in an earlier stage of development. Decisions which have a wide impact should not, ideally, be highly error-prone. Error-prone decisions should be followed as soon as possible by consideration of anything which may invalidate them. Decisions about implementation should be taken after decisions about specification.

From this point of view, it seems clear that 'top-down' and 'stepwise refinement' methods are to be studiously avoided. Using such a method, the developer begins by deciding the top-level structure of the system, decomposing it into its largest constituent parts. Undoubtedly, this is the worst possible decision to place at the beginning of development. It is a decision about the system itself, which, *ex hypothesi*, is not yet well-known: it is therefore highly error-prone. It has the widest impact of all those decisions within the developer's discretion, because it sets the context for all later structural decisions: if the top-level structure is wrong, it will not be easy to salvage much of the work done on lower levels. And, finally, if this first decision is wrong, it may not be invalidated until late in the development,

We may suspect that developers who claim to be using top-down or stepwise refinements methods are, in fact, doing something quite different. The real work of development is done, informally and invisibly, in the developer's head, where something approximating to an outline of the complete system is visualised. This outline is then documented in a top-down fashion, and enough details filled in to complete the work. A brilliant, or even highly competent, developer may be able to work quite effectively in this manner, but its limitations are obvious [11].

6 Maintenance and Complexity

Many kinds of software system — perhaps most kinds — must be readily adaptable to changes in their specifications. We call this adaptation 'maintenance', a usage which is unique to software engineering. In other branches of engineering, the word 'maintenance' usually means the activity of guarding against or repairing the physical degradation which afflicts the product. Bridges must be painted to avoid corrosion; resistors must be examined and replaced if their value has strayed outside the specified tolerance; the oil in the engine sump must be drained and replaced; the potholes in the road must be filled in. Analogous activity is needed in software systems of some kinds: a database system may need periodic examination to detect and repair errors in chaining between records; an index sequential file may need reloading when too many insertions have reduced the access speed or filled the overflow area; a partitioned data set may need to be reorganised to make dead space available for new members. But we do not usually mean such things by ‘maintenance’: we mean the activity of changing the system to satisfy changed specifications, much as the design of an aeroplane may be changed to give a ‘stretched’ version, or a bridge may be widened to carry a greater load of traffic.

It is well known that maintenance in this sense accounts for a large part of the expenditure on data processing systems. This need not be a cause for dismay or concern: we might congratulate ourselves because our products are so adaptable that our customers’ needs can be satisfied at the lower cost of maintenance rather than the higher cost of complete redevelopment. But few software engineers, and even fewer customers, would join in the congratulations. The cost of maintenance is generally considered to be too high, in the sense that comparatively small changes in specification often require very difficult and expensive changes to the system. We might also consider the cost of the maintenance that is not carried out because it is too difficult or expensive for the customer to accept it at the offered price. We rarely get the opportunity to measure the cost of non-maintenance.

The source of excessive maintenance cost is disparity between the structure of the specification and the structure of the system. The change to the specification may be small, simple, and local; if the consequent change to the system is large, complex, and diffuse, that must mean that the structure of the system is different from the structure of the specification. Often, the system will also be excessively complex in itself, making any change difficult and even dangerous.

Our chief tool for mastering and avoiding complexity is the ability to view a system as a relatively small number of relatively large parts, connected in a clear and simple way. If the cost of maintenance is to be low, then the parts and connections in the system must correspond closely to the parts and connections in the specification. At the same time, the parts and connections in the specification must correspond closely to the parts and connections in the customer’s view of the problem domain. A fundamental difficulty in achieving this goal is the inevitable difference between the structure of a good specification and the structure of an efficient implementation. To return to the earlier example of the payroll system, the specification should be structured so that one of its distinct parts is a statement of what happens to one employee during the whole period of his employment; but an efficient implementation may require to contain a distinct part which is a weekly batch program dealing with the events that have affected all current employees during the past week. We cannot avoid the problem of mediating between these two different structures; the mechanised conversion from process to procedure form is a significant component in a solution to this problem. To the greatest possible extent, we should aim to take advantage of this tractability of the software medium, of the ability to transform one piece of software into another of related but different structure [12].

7 Standard Products and Parts

Most engineering products are highly standardised, and their designers work within narrowly defined bounds. Each product falls into a well-known type, and has a readily recognisable structure; the engineer is not expected to produce a revolutionary design, but rather a carefully crafted set of choices within the accepted parameters. The introduction of a significantly different structure for a product is a revolution, and regarded as potentially dangerous. After countless suspension and cantilever bridges has been designed and built, it was a daring engineer who conceived the box girder bridge. After sixty years of reciprocating internal combustion engines, it was a daring innovation to produce a car powered by a rotary engine.

There are many reasons for this high degree of standardisation. One important reason is the visibility of the physical product, which we have referred to previously. The automobile manufacturer who first abandoned the separate chassis and body and adopted the integral design could hardly hope to keep the advance a secret from his competitors. Another reason is the widespread use which most physical engineering products receive. Huge numbers of people use automobiles of a particular design, or cross a particular bridge on their way to work, or ride in elevators of a particular design. And they use other automobiles and bridge and elevators, too. So there is a strong tendency towards uniformity in customer's expectations of a particular class of products. This tendency is much less marked in software, where many products are created for customers who have no experience of other similar products to guide their evaluation. Where usage of a class of product is widespread, as of Fortran compilers, there is a stronger tendency toward uniformity of expectation and hence of product.

Yet another reason for standardisation of products is the use of standard parts. The manufacturer of a physical product is usually forced to buy many or even all of the parts he uses; these parts are bought from component companies who supply identical parts to the other manufacturers. In software, the developer always has the option — even if it is not necessarily the best option — of building everything in-house. At the 1968 NATO conference, M. D. Mcllroy, advocating the creation of a software components industry, complained: [13]

"When we undertake to write a compiler, we begin by saying 'what table mechanism shall we build?' Not 'what mechanism shall we use?', but 'what mechanism shall we build?' I claim we have done enough of this to start taking such things off the shelf."

A dramatic illustration of the way standardisation comes about is provided by the microcomputer industry. Where discussion about standard machine architecture had limped along for ten years in the mainframe and minicomputer industry, the microcomputer industry has standardised itself immediately, simply because very cheap standard CPU chips are available which the manufacturer can not produce in-house.

This has scarcely happened in the software industry. There are a few examples of standard components or packages, such as libraries of mathematical routines, that were already visible in 1968. But remarkably little else. We may perhaps attribute the lack of standard components to both economic and technical factors. If a putative standard component is small, the cost of building it in-house may be no greater than the cost of identifying and obtaining the required item from the supplier. If it is larger, its specification is likely to be larger, and there is correspondingly less likelihood of finding what is needed in a supplier's catalogue. The technical factors are associated with the interface specification. Traditionally, the general-purpose software components contributed to an installation's program library by hopeful authors have been cast in the form of procedures. The procedure interface is very satisfactory for mathematical functions, but not for much else. The interface specified for the library routine always looks to the potential user to be arbitrary, difficult to understand, and impossible to reconcile with his existing design; so he writes his own version to his own interface specifications. We may note that one environment where general purpose software does seem to be widely used is the UNIX environment. Not surprisingly, a typical UNIX component is a sequential process communicating by message streams (pipes) with other processes; the message stream interface, for a wide class of application, is both convenient and well-suited to the definition of standard software components.

8 Some Conclusions and Suggestions

The use of the term software engineering expresses an aspiration, not a fact. The established branches of engineering are old; we are young. Their work is organised into well-defined specialisations; ours is still largely ill bounded and undifferentiated. Their products are standardised wherever possible; ours are most often built *ad hoc*. They examine and evaluate their products; we spend more time contemplating our navels. They have components industries; we do not. Their disastrous mistakes make headlines in the newspapers; ours are too commonplace to merit remark.

Many of these differences, and many of our difficulties, flow from the intangible nature of the software product. It is hard for us to draw boundaries between different development stages and activities; it is hard to constrain the engineer's choices enough to make his work manageable without preventing him from building an efficient product. But this intangible nature of the product is our greatest advantage, and we have derived too little benefit from it. We can manipulate, rearrange, reconfigure, and transform a software product at a very low cost, by using the computer itself; we have not done so as we should.

The crux of the matter is the relationship among three structures: the structure of the problem in the problem domain; the structure of the written specification; and the structure of the implementation. For a large class of problems, including many in data processing, process control, message switching, and embedded applications, the problem domain is sequentially ordered in time: the natural specification structure to capture the essence of the problem in its context is that of a set of communicating sequential processes. Where the number of these processes, their elapsed execution times in the real world, and the densities of their activities are well-fitted to the available machine, operating system, and programming language, the development of the system can proceed reasonably smoothly. The ideal development here is one in which the specification processes are programmed directly in a process-orientated programming language, and the resulting program can be executed directly on the machine.

But often, especially in data processing applications, the configuration and dimensions of the specification process set are ill-fitted — even to the point of incompatibility — to the programming language and to the machine and its operating system. Present operating systems are conceived for the execution of procedures (whose execution is, conceptually, instantaneous), or of small sets of processes with short execution times and dense demands for machine cycles. A specification with 10,000 processes, each taking 50 years to execute, and demanding only 100 seconds of machine time over the 50 years, simply does not fit the machine. Traditionally, the incompatibility has been resolved by choosing between Scylla and Charybdis. Either the specification is cast in a problem orientated form which the customer can hope to understand, whereupon the structure of the design will bear no visible relation to the structure of the specification: or the specification is cast in design-orientated form, whereupon the structure of the specification will bear no visible relation to the structure of the problem in its domain.

We should cast the specification in the problem-orientated form (of course!), and we should derive the design structure (that is, the implementation structure) by systematic transformation. The explicit, process-structured, specification is itself a fully detailed executable text. But it needs to be rearranged and transformed before it can be executed efficiently. The computer should be the essential tool in this activity. The transformations should be carried out mechanically, so that the correctness of the specification is preserved; but they should be chosen by the engineer, interacting with the transformation system. Some transformations of the appropriate kinds have been studied and described; some have already been mechanised [14, 15]. But there is a long way to go before use of such transformations becomes widespread and well-supported.

One benefit that might accrue from this approach to software development is a separation of languages. Our present languages are disgracefully complicated, as they must inevitably be, serving simultaneously the incompatible purposes of specification and implementation. In place of a programming language, we might have a pair of related languages: a specification language, ruthlessly purged of implementation features; and a language for directing the transformation of specifications into efficient implementations. Each language of the pair could be much simpler than today's programming languages. Along with this separation of languages would come a separation of development activity along more rational and intelligible lines. It would be easier to determine a boundary between the specification engineer and the implementation engineer, and fruitful specialisation might develop both within an organisation and between one organisation and another. Perhaps specialisation is both the precondition and the hallmark of a mature discipline of engineering.

References

[1] *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE*; ed P. Naur and B. Randell; 1969.

[2] *The Preparation of Programs for an Electronic Digital Computer*; M. V. Wilkes, D. J. Wheeler and S. Gill; AddisonWesley, 1951. Quoted in *A History of Computing in the Twentieth Century*; ed N. Metropolis, J. Howlett and G-C. Rota; Academic Press, 1980.

[3] *Hierarchical Program Structures*; O-J. Dahl and C. A. R. Hoare; in *Structured Programming*; O-J. Dahl, E. W. Dijkstra and C. A. R. Hoare; Academic Press, 1972. See also *Design of a Separable Transition-diagram Compiler*; M. E. Conway, Comm ACM July 1963.

[4] *Cooperating Sequential Processes*; E. W. Dijkstra; in *Programming Languages*; ed F. Genuys; Academic Press, 1968.

[5] *Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept*; W. M. Gentleman; Software Practice and Experience, May 1981.

[6] *Communicating Sequential Processes*; C. A. R. Hoare; Comm ACM, August 1978.

[7] *The Design of Data Type Specifications*; J. V. Guttag, E. Horowitz and D. R. Musser; in *Current Trends in Programming Methodology*, IV; ed R. T. Yeh; Prentice-Hall, 1978.

[8] *Structured Design*; E. Yourdon and L. L. Constantine; Prentice-Hall, 1979.

[9] *Structured Programming*; R. C. Linger, H. D. Mills and B. L Witt; Addison-Wesley, 1979.

[10] *Software Development: A Rigorous Approach*; C. B. Jones; Prentice-Hall, 1980.

[11] *Programming as a Cognitive Activity;* T. R. Green; in *Human Interaction with Computers*; ed H. T. Smith and T. R. Green; Academic Press, 1980.

[12] *Information Systems: Modelling, Sequencing and Transformations*; M. A. Jackson; Proc 3rd international Conference on Software Engineering, 1978.

[13] op cit [1].

[14] *Some Transformations for Developing Recursive Programs*; R. M. Burstall and J. Darlington; Proc International Conference on Reliable Software, 1975.

[15] *A System for Developing Programs by Transformation*; M. S. Feather; PhD Thesis, University of Edingburgh, 1978.

Angewandte Informatik 2/8 pp96-103 0013-5704/82/2 0096-08S 02.00/0