Problems, Methods and Specialisation

Michael Jackson

(A Contribution to the Special Issue of SE Journal
on Software Engineering in the Year 2001)

Abstract

Software Engineering is not a discipline: it is an aspiration, as yet unachieved. Many approaches have been proposed, including reusable components, formal methods, structured methods, and architectural studies. These approaches chiefly emphasise the engineering product: the solution rather than the problem it solves. An approach to understanding and classifying software development problems in terms of *problem frames* is suggested. In addition to such general approaches, specialisation is essential: the established branches of engineering are all specialisations. Some specialisations have arisen in software development, notably in compiler construction and software for personal computers. More are needed.

1 The Aspiration

The term *Software Engineering* is usually thought to date from 1968. The report of the first Software Engineering conference [13] explains the background:

> "In late 1967 the [NATO Science Committee] Study Group recommended the holding of a working conference on Software Engineering. The phrase 'software engineering' 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."

The aspiration was for software engineers to take a merited place among the ranks of civil, electrical, aeronautical, chemical, structural and automotive engineers and their colleagues in the other established branches. By the year 2001 this aspiration, to create and practise a discipline of Software Engineering, will be 33 years old—one third of a century. But the 26 years that have passed so far give little reason to think that the aspiration will have been achieved in another seven years. Software engineering has not become like the other established branches, and it will not become so in the near future.

One reason is simply that it's difficult. Fred Brooks, in his much-quoted paper [2], said:

> "The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract, in that the conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed.
>
> "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, ...
>
> "If this is true, building software will always be hard. There is inherently no silver bullet."

Another reason for our failure is that we have added to the essential difficulty of the task by a simple lack of professionalism. In its earliest days, software development grew out of mathematical programming, whose practitioners were mathematicians, astronomers and physicists. They regarded computer programming as an essentially trivial activity by comparison with their professional work. A boundary between 'user programming' and 'professional programming' eventually grew up in the mainframe Fortran and COBOL culture; but later it became blurred by the emergence and eventual hegemony of the PC culture. Visual Basic, Paradox and Lotus 1–2–3 are all tools intended for both the most serious and the most casual use. The established branches of engineering do not suffer in this way: there are no casual builders of motor cars or bridges; there is no do-it-yourself kit for designing steel structures or aero engines. But in software development it is not easy to draw a clear line between the casual developer and the serious, professional, developer.

As a result, although some efforts have been made to professionalise and regulate software engineering, following the pattern of nineteenth-century regulation of civil, railway, and electrical engineering, they have not yet borne edible fruit. Software development is still largely an amateur activity in a very important sense. In a paper analysing the Therac-25 accidents [11], an official of the US Food and Drug Administration (FDA) is quoted as saying:

> "A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering."

The software for the Therac-25 had errors in simple sequential logic. Its developers were also apparently unaware of the pitfalls associated with concurrent access to shared variables, a problem whose solution, by T Dekker, had been reported in the year of the NATO conference [4].

Another reason for the failure of our aspiration to become engineers is that many theorists and practitioners, even among the most professional, are naïve. The continuing search for panaceas, for universal materials out of which everything can be made, for universal methods to solve all problems, strongly suggests that we have not yet begun to understand the nature of our field. Object-orientation is only the most recent in a long line of proposed panaceas, all claiming unbounded applicability and potency. Even those who complain most bitterly of our lack of professionalism are inclined to exhibit this naïveté. Proponents of formalism, for example, are right to claim that if software development were carried out with more care, more mathematical rigour, and more precision, some of the egregious errors that are so often made would probably be avoided. But the prescription is too readily offered as a complete and universal cure. It is not. Our diseases lie deeper.

2 Prescriptions for Success

Many prescriptions have been put forward to cure our obviously unhealthy state. One strong medicine is the development and use of catalogues of ready-made components, following the example of electronic engineers and some others. In an invited address at the NATO conference [13], Doug McIlroy said:

> "Components, dignified as a hardware field, is unknown as a legitimate branch of software. 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."

Some progress towards this goal has been made in particular environments: libraries of mathematical routines and object classes for GUIs are notable examples. Programming languages come equipped—like C—with libraries of callable functions, or—like COBOL—with repertoires of elaborate statements. And in many areas there is a steady drain of functionality from application programs into the technical environment—the operating system and the DBMS and the communications system. Yet most everyday programming still defies reduction to component assembly. Either the need is too specialised, and the interfaces and functionality of the available components don't fit the context in which they are to be used. Or, perversely, it is too general. How many programmers, even while you read this sentence, are programming linear search? And some of them, no doubt, are programming it wrongly.

Proponents of formal methods, such as Z [20] and VDM [10], offer a different diagnosis and therapy: they point to the often neglected mathematical aspect of software development. Undeniably, computer programs can be regarded as formal things, and can therefore be subjected to mathematical treatment. Formalists want to emulate the practitioners of the established branches of engineering. The structural engineer calculates stresses; the automotive engineer calculates torques and accelerations and wind resistance. The software engineer, they believe, should be calculating too. Calculation would reveal the implicit precondition of an operation, and show that it might be invoked when its results were unpredictable. Or it might show that a proposed refinement was invalid. Obviously, it is absurd to tolerate vagueness, confusion and uncertainty where precise calculation is possible. The Roman engineers would surely not have refused the offer of a better system of arithmetic.

But in most areas of software development the prerequisites for a more mathematical approach are not yet in place. Engineers in the established branches make their calculations in very well-defined and narrow contexts.

An automotive engineer, for example, does not set out to design a new car *de novo*, calculating whether to use a steam engine, whether to use tracks, or articulated legs, or four wheels, or eight, or whether the driver should face sideways or forwards. There is a repertoire of standard designs, evolved over many decades of experience, and each new design is at most a small perturbation from the standard. To combine the chassis and body in one unit, or to set the engine transversely, was a radical departure open only to the most brilliant and daring designers. In practice the design space is very narrow. The engineering calculations are done on familiar components in a familiar configuration.

In most areas, software development has no such established standard designs. The structures offered by formal methods are too general to narrow the design space. The system is seen as a set of operations on a state, or as an assemblage of communicating sequential processes: these are general computational paradigms, and offer almost no guidance to the designer faced with any particular development task. Understanding the problem and structuring the solution is the primary need. Only when that has been done, at least in outline, is the subject matter for calculation available. Rigorous description and calculation must come second. It may then be crucially important for some aspects of the task, but it is not always at the heart of the problem.

'Structured' [21] and 'Object-Oriented' [12] methods are concerned with the problem of designing software structures. In most cases their prescriptions, too, are excessively general. Every piece of software can be described—at least, to some extent—by a dataflow diagram. Every program can be built—more or less appropriately—in an object-oriented language. The methods associated with these notions give little or no guidance in the crucial decisions facing the designer: *What* objects are needed? *What* functions and data streams? The material in which the design is to be expressed gains generality from its recursive structure: the function processes in a dataflow diagram can themselves be described by dataflow diagrams; and a subclass in an object inheritance hierarchy can itself be a superclass. But this very generality gives a design space that is too wide for comfort.

3 Problems and Solutions

Garlan and Shaw, in their architectural approach [5], move towards a narrowing of the design space facing the developer. They are concerned to classify and study common architectural styles such as pipes-and-filters and blackboard-connected-processes. Their approach has something in common with the work on programming clichés [17], and with the 'Pattern Language' approach [8] that draws on the work of the software methodologist's favourite architect, Christopher Alexander [1].

All of this work approaches the design task from the side of the software-hardware machine. It focuses on the characteristics and structure of the solution, and so offers a prospect of evolving the kinds of standard design that are found in the established branches of engineering. But it tends to ignore the complementary—and perhaps more important—side of the development task: the characterisation of the problem to which the software-hardware machine provides a solution.

Johnson, a leading proponent of patterns, says in [9]:

> "Alexander focuses as much on the problem to be solved and the various forces on the problem as he does on the solution to the problem. We have a tendency to focus on the solution, in large part because it is easier to notice a pattern in the systems that we build than it is to see the pattern in the problems we are solving that lead to the patterns in our solutions to them."

Johnson's point is perceptive. The concentration on solutions is more widespread than may be recognised. Some methods claim to be analysing or structuring the problem when in fact they place all their emphasis on describing a solution. Johnson gives one reason: solution structures are more easily comprehended than problem structures. A programming language and environment inevitably provides concepts and terms—procedures, processes, data structures, functions, types, messages, invocations, files—that come readily to hand when we want to describe the shape of a software solution. Problem environments are less helpful and suggestive. Another influence diverting attention from problems is that a software system is often a partial simulation of its problem domain: the developer is then easily persuaded that a description of the software *is* a description of the problem. But it is scarcely more so than a description of a car steering wheel is a description of the physiology of the driver's arm and hand.

Focus on the problem to be solved is implicit in the established branches of engineering, because the materials and techniques, and the repertoire of standard designs, have a narrowly limited applicability. An automotive engineer expects to design motor cars, and would be at a loss if set to design a bridge. But the materials and techniques of software development, in those areas where we lack standard designs, have a very wide applicability. The problem can not be taken for granted: it must be quite explicitly identified and analysed.

4 Problem Contexts

Responding to Fred Brooks, Wlad Turski [19] pointed out that software development is concerned with more than a formal computing system:

> "But, as we have observed before, software has another aspect: that of describing properties of an application domain. In this sense software does not always relate two formal systems [software and hardware]. Many computer application domains are not formal systems at all.
>
> "There are two fundamental difficulties involved in dealing with non-formal domains (also known as 'the real world').
>
> 1 Properties they enjoy are not necessarily expressible in any single linguistic system.
>
> 2 The notion of mathematical (logical) proof does not apply to them."

The point is that software developers can not ignore the application domain: identifying, capturing, understanding and analysing the problem in its context is an integral and essential part of our concern. In consequence, software development—unlike hardware development—is about banking and telephone switching and air traffic control and avionics and compiling programs and calculating spreadsheets and formatting and displaying texts. In short, it is about everything that can furnish the subject matter for a program or system.

At first sight, this might be taken to mean that a software developer must be—or become—expert in the application domain. But this is not so. What it means is that the software developer must become expert in those aspects of the application domain that affect the design and construction of the software. A domestic heating engineer is not expected to be an architect, but must be expert at analysing those aspects of a building—the ambient climatic conditions, the locations of doors and windows, room volumes, wall and roof insulation, and traffic inside a house—that affect the demand for heat and present opportunities and difficulties to the designer of a domestic heating system.

For a software developer, the important aspects of an application domain are those that determine how easy or hard it will be to *describe* the domain, and, where necessary, to *simulate* it or to exert *control* over it. The software developer must therefore be skilled in *phenomenology*—recognising and capturing the significant elementary phenomena of the domain and their relationships; in the *technology of description*—the techniques of capturing difficult properties such as causality, and of constructing a description in parts such that each one says as much as is necessary, with perfect clarity, but no more; and in *formalisation*—choosing and expressing the abstractions and generalisations necessary to bring an informal reality under sufficient intellectual control for a particular purpose. The software developer's expertise in a domain [7] is, above all, an active understanding of how to describe it. The products of this expertise provide the foundation on which a system's requirements and specifications may be built.

Some attention [16] has been paid to application domains considered generically: to *the banking domain*, or to *missile applications*, or to *strategic-management-support-systems*. And some development methods (for example, JSD [6]) have advocated explicit analysis and description of the particular 'real world' for each particular system. Identifying and understanding the 'real world' must be a first step towards understanding the problem to be solved. It is the *context* in which the problem exists.

Consider, for example, this classic problem (adapted from [18]):

> "A patient-monitoring program is required for a hospital. Each patient is monitored by an analog device that measures factors such as pulse, temperature, blood pressure, and skin resistance.

> "The program reads these factors on a periodic basis (specified for each patient). For each patient, safe ranges for each factor are specified. If a factor falls outside a patient's safe range, or if an analog device fails, the nurse's station is notified."

The problem context is a real world, separable into distinct domains (using the word to denote a distinguishable part of the specific problem context rather than generically to denote a class of problem context). There are patients; there are analog devices; there is a source of specifications (presumably the medical staff) of safe ranges and monitoring periods; and there is a nurse's station. The analog devices are connected to the patients; the nurse's station is not connected to any other part (but will be connected to the machine we are to build). The medical staff will deliver their specifications of ranges and periods to the machine in some way. All of these domains will demand careful investigation and description. Ultimately, of course, the problem will be to construct the machine. In software development, the problem is always to construct a machine—but it must be a machine that will fit into a particular context in the world, where its costs and benefits will be felt and evaluated.

5 Problem Frames

Identifying the context of a problem is only the first step towards understanding the problem itself. The ancient Greek mathematicians paid a lot of attention to the study of problems, treating it separately from the related study of solutions and solution methods. An admirable little book by Polya [15] gives an account of their ideas. They classified mathematical problems into *problems to find or construct*, such as:

> Given lengths a, b, and c, construct a triangle whose sides are of those lengths.

and *problems to prove*, such as:

> Prove that if the four sides of a quadrilateral are equal then its diagonals are mutually perpendicular.

A problem can be characterised by its *principal parts* and a *solution task*. The principal parts of a problem to prove are the *hypothesis*—that the four sides of a quadrilateral are equal; and the *conclusion*—that its diagonals are mutually perpendicular. The solution task is to show that the *conclusion* follows from the *hypothesis*. The principal parts of a problem to construct are the *unknown*—a triangle; the *data*—three lengths *a, b,* and *c*; and the *condition*—that the triangle's sides are of the lengths given in the *data*. The solution task is to construct the *unknown* so that it satisfies the *condition* with respect to the *data*.

The essence of the principal parts is that they are parts of the problem, not of a solution or of the steps towards a solution. This allows a discussion of methods to be cast in the appropriate terms: that is, in terms of the particular kind of problem to be solved. Polya gives methodological recommendations for each kind of problem. For problems to prove he suggests: 'Try to think of a familiar theorem having the same or a similar *conclusion*'; 'Ask whether the *conclusion* is more likely to be true or false, given the *hypothesis*'; 'Consider what other *conclusions* follow from the *hypothesis*'; and so on. For problems to find or construct he advises: 'Split the *condition* into parts'; 'Check that you are using all the *data*'; 'Think of a variation of the *unknown* to bring it closer to the *data*'; 'Ask whether the *condition* is sufficient to determine the *unknown*'. Naturally, the recommendations are different for the two kinds of problem, because they are cast in terms of their different principal parts. A strong problem-solving method must be closely constrained by the properties of the problems it purports to solve.

The principal parts and the solution task of a problem, then, form a structure within which the problem can be considered systematically, and an appropriate solution method chosen or devised. Such a structure may be called a *problem frame*. To understand a problem is to have fitted it into an appropriate problem frame by identifying its principal parts and the solution task. This is the central activity in problem analysis. Even in the small problems discussed by Polya, the choice of problem frame and the identification of the principal parts is not always obvious. As Polya points out, the problem 'Prove that there is an infinity of primes' does not readily fit the pattern of problems to prove; and the problem 'Show that there is at least one prime between 6 and 10' can be treated either as a problem to find or as a problem to prove.

The idea of a problem frame can be applied quite directly in software development. We must begin, of course, with the problem context, identifying the relevant parts of the real world and making an initial assessment of their properties and relationships. In mathematics we can summon up a rich body of domain knowledge by a single word. When we say that the *unknown* is a triangle, we expect the reader to know what that means without further explanation. We also expect the reader to be familiar with at least some of the properties of a triangle—that the sum of the interior angles is 180 degrees, that the length of each side must be less than the sum of the lengths of the two other sides, and so on. But in software development, the 'real world' domains are much richer and more varied than the abstract domains of mathematics, and their properties must be made explicit by the hard work of careful description. As Turski points out, they are often non-formal and their properties may not be expressible in any single linguistic system. That is why the software developer must be technically well-equipped to investigate the relevant properties of these 'real world' domains and formalise them in a variety of appropriate languages.

6 Software Problem Frames

Polya's two problem types, as he recognises, are insufficient to structure all problems even of the small class he discusses. Software development problems are far more various, and we must expect to need many more than two problem frames. But few problem frames have been explicitly described: methods tend to deal in solutions rather than in problems, or to leave the problem frame partly hidden, implicit in the terminology used to describe the method. Here we can only identify a few software problem frames and describe them roughly. We may call them the JSD frame, the Workpiece frame, and the Environment-Effect frame.

6.1 The JSD Problem Frame

The JSD method [6] uses a problem frame appropriate to the development of information systems. The principal parts of the JSD problem frame are these:

Real World This is the particular world about which the system is to produce information. It is a domain in the problem context. It is dynamic: that is, it has a behaviour over time, in which events, and consequent state changes, occur. The *Real World* is *autonomous*: that is, its events and state changes are regarded as occurring spontaneously and not as externally stimulated or controlled. In particular, they can not be controlled by the machine to be built: for the developer using this problem frame, the *Real World* is not to be changed or constrained, but is merely the subject of observation and reporting.

Information Outputs These are the outputs containing the required information about the *Real World*. They are a domain in the problem context.

Requests These are the information requests made by the users of the information system. They are a domain in the problem context. The *Requests* are an unstructured stream of time-ordered events. The users are treated purely as a source of this unstructured stream: for example, individual users are not distinguished, and the domain is not regarded as having any internal state.

System This is the machine to be built. It is connected to the *Real World*, *Information Outputs*, and *Requests* domains. It produces the *Information Outputs* both in response to the *Requests* and autonomously according to the state and behaviour of the *Real World*.

Function This is the required relationship that the *System* must cause to hold among the *Real World*, the *Information Outputs*, and the *Requests*. Some *Information Outputs* are to be produced when certain events occur, or certain conditions hold, in the *Real World*, others in response to *Requests*.

The Solution Task in the JSD frame is to construct a *System* that models, or simulates, the *Real World* and satisfies the *Function*.

6.2 The Workpiece Frame

Another problem frame may be called the Workpiece frame. It regards the machine as a production tool for textual or graphic documents. It may be suitable for such applications as simple word-processing. It has these principal parts:

Workpieces The objects, often textual documents with embedded graphic or tabular parts, that are to be worked on. The *Workpieces* are an intangible domain of the problem context. They are capable of changing their state, but only as a result of external action: they have no autonomous behaviour.

Operation Requests The requests made by the users of the system for operations to be performed on the *Workpieces*. They are a domain of the problem context. Like the *Requests* of the JSD frame, the *Operation Requests* are regarded as an unstructured stream of events, in which neither individual users nor domain states are distinguished.

Operations The operations that the users can ask the *Machine* to perform on the Workpieces. They are a required relationship between the *Operation Requests* and the states of the *Workpieces* domain.

Machine The machine to be built. It contains a reification of the *Workpieces*, and performs the *Operations* on them in response to *Operation Requests*. It has no autonomous behaviour.

The Solution Task in the Workpiece frame is to construct the *Machine* to perform the *Operations* on the *Workpieces* in response to the *Operation Requests*.

6.3 The Environment-Effect Frame

A final example may be called the Environment-Effect frame. It has something in common with an approach described by Parnas and Madey [14]. It is suitable for an embedded system that controls an external domain. The principal parts are:

Environment The domain to be controlled. It has state, and a behaviour that is partly autonomous and partly responsive to externally caused events.

Connection The connection between the machine to be built and the *Environment*, by which the machine can sense and affect states and events in the *Environment*. It is a domain of the problem context.

Machine The machine to be constructed. Its behaviour must be partly autonomous and partly responsive to events and state changes of the *Environment*.

Requirement The domain properties and behaviour—relationships among phenomena of the *Environment*—that the *Machine* is to bring about and maintain.

The Solution Task in the Environment-Effect frame is to construct the *Machine* so that it senses and controls the *Environment* through the *Connection*, and ensures satisfaction of the *Requirement*.

7 Frames and Methods

Although these problem frames have been only roughly sketched, it should be evident that they are far from interchangeable. The chosen problem frame must fit the problem. The domains of the problem context must have the characteristics demanded of corresponding principal parts, and must be connected, directly or indirectly, to each other and to the machine in the way demanded. The frame must have appropriate parts to accommodate all the required properties and relationships to be described and considered.

The Workpiece frame, therefore, would be useless for developing a system to control a chemical plant, because the Workpieces domain is assumed to be intangible and inert, but the chemical plant is neither. The JSD frame is inadequate for an embedded system: its Real World domain is assumed to be autonomous, whereas the environment of an embedded system is to be controlled by the machine. So the JSD frame has no principal part like the *Requirement* of the Environment-Effect frame, to capture and accommodate the domain properties that are *desired* as opposed to those that are *given*.

The principal parts of a problem frame furnish the material for a development method. A method prescribes a problem frame, and offers guidance on identifying the domains of the problem context and the principal parts of the frame. It stipulates that certain descriptions are to be made, starting with descriptions of principal parts. It may stipulate an order of description; the various languages to be used; and operations—such as abstraction, composition, transformation, decoration and refinement—by which new descriptions may be derived from those already made. The culminating descriptions, of course, are descriptions of the machine to be built.

A good software development method prescribes a very specific problem frame, and exploits its properties to the full. The known characteristics of each principal part allow an appropriate language to be chosen for its description. For example, the *Real World* in a JSD problem can be described in terms of concurrent sequential processes expressed as regular grammars in a diagrammatic representation. Known relationships among the domains of the problem context allow descriptions to be composed in simple ways. For example, the *Workpieces* of the Workpiece problem frame can be described as instances of an abstract data type, and the *Operations* are then simply the operations of the type. Potential difficulties can be categorised and specific solutions offered. Techniques and notations for description can be sharply honed to fit the characteristics of the principal parts to be described and their significant relationships. The closer the problem frame fits the problem, and the more insistently the method exploits the properties of the frame and its principal parts, the more powerful the method can be in identifying and overcoming obstacles to solving the problem. A *simple* problem is a problem for which we have a close-fitting frame and an effective method that exploits it.

8 Complexity and Composition

But software development problems are too various and too rich to be captured by any reasonably small set of problem frames. They exhibit *complexity*, in the sense that more than one problem frame is needed for each problem. The construction of a CASE tool might seem to fit comfortably into the Workpiece problem frame. But if information is also needed about the progress of the work done using the tool, both the Workpiece problem frame and the JSD frame must be used. The Workpiece frame is used for building the basic functionality of the tool, and the JSD frame for generating the information about the progress made by the tool's users. Two views must be taken of the one problem, two methods must be applied, and two solutions must be composed. And if the CASE tool must also impose method constraints on its users, the Domain-Effect frame must be applied too. The *Workpieces*, *Requests*, and *Operations* then constitute the *Environment* of this further frame. Even the classic patient-monitoring problem mentioned earlier has a complexity. It might be viewed as a JSD problem (with the addition of the analog devices as a *Connection* interposed between the *Real World* and the *System*). But the need to detect and report failure of the analog devices suggests that they must be treated as the *Real World* in a second instance of the JSD problem frame.

These complexities, of course, are purely relative to the available repertoire of problem frames and associated methods. Recognising and resolving such complexities is the essence of decomposition of problems into sub-problems. But it is decomposition guided by a clear idea of what constitutes a sub-problem: it is a problem for which an appropriate problem frame and an effective associated method are known.

Traditionally, notions of decomposition have stood as if they were self-sufficient. But they are not. If a problem has been decomposed into sub-problems, then the resulting sub-solutions must be recomposed into one solution. Having divided to conquer, we must re-unite to rule. This consequence of decomposition is less apparent when solutions are decomposed within a single computational paradigm. Procedure call allows decomposition into a hierarchy of unlimited depth and width; message-passing allows decomposition into an unlimited number of objects and instances. The subsequent composition is entirely straightforward, within the paradigm.

But decomposition according to more specific problem frames and methods poses a significant composition problem. The same domain—for example, the Workpieces—appears as two different principal parts in two different problem frames, where the associated methods may exploit or rely on different characteristics. There is then a potential difficulty in implementation, to give one domain within the machine an implementation that conforms to both sets of characteristics. This kind of composition exploits what traditional engineers call *The Shanley Principle*, whose relevance to software development was pointed out by de Marneffe [3]:

> " ... If you make a cross-section of, for instance, the German V-2 [rocket], you find external skin, structural rods, tank wall, etc. If you cut across the Saturn-B moon rocket, you find only an

external skin which is at the same time a structural component and the tank wall. Rocketry engineers have used the "Shanley Principle" thoroughly when they use the fuel pressure inside the tank to improve the rigidity of the external skin!"

9 Specialisation

This is the essence of specialisation. The specialist concentrates on problems of a relatively small class, and on the techniques for understanding and solving them. That means being familiar with typical problem contexts of the class, and skilled in analysing and describing domains of the kind that are usually found there. The specialist is also familiar with a set of problem frames that is usually sufficient to cover any problem of the class closely and completely, and with the identification of their principal parts with the context domains. The specialist's constructive skill lies chiefly in composing the solutions to sub-problems that have been identified and fitted into problem frames of the familiar set. This sharp focus allows a body of specialised knowledge to be built up, and the appropriate 'theoretical foundations' and 'practical disciplines' to be brought to bear in a familiar and very specific context.

This is what traditional engineers have done, and is the origin of the established branches of engineering. The automotive engineer commands the set of problem frames that cover the design of an automobile, and the techniques for fitting their solutions together to give a well-designed car. The large aspiration to place the whole of software development alongside the established branches as one more branch of engineering is misconceived. Our aspiration should be rather to develop specialised branches of software engineering, each meriting its own place alongside the specialised established branches.

There are already some examples of such specialisation in software. Compiler writing is one notable example. Difficulties specific to the problem area, such as context-sensitive grammars, are recognised and classified. Description transformations, such as the derivation of LALR parsing tables from grammars, are part of a standard repertoire. The broad structures of different functions of a compiler, such as lexical and syntactic analysis, code generation, and peephole and global optimisation, are studied, and ways are devised of composing them into efficient products.

There are pressures to specialise in other software fields. The production of shrink-wrapped software for personal computers is especially subject to strong pressures of this kind. Magazines review competing products, and so lead to a kind of self-interested cooperation among producers as they study each other's products and try to emulate the good features offered by their competitors. Users add a strong pressure towards standardisation, both by demanding interoperability and by insisting on the kind of 'look and feel' similarity that makes it possible for a competent car driver to get into almost any car and drive it immediately.

But in many fields, where each software project is unique, these pressures are not felt. It is an urgent challenge, if we aspire to the status of engineers, to identify, study and embrace the specialisations into which our discipline must be divided. Only specialisation, and the problem-oriented understanding that must guide it, can advance software engineering from an amateur to a professional activity.

10 Acknowledgements

Several people—John Dobson, Daniel Jackson, Ralph Johnson, and Wlad Turski—gave me many valuable comments on an earlier version of this paper. I have been greatly helped by their interest and advice, and I thank them all.

References

[1] Christopher Alexander, Sora Ishikawa and Murray Silverstein; A Pattern Language; Oxford University Press, New York, 1977.

[2] Frederick P Brooks Jr; No Silver Bullet—Essence and Accidents of Software Engineering; Information Processing 86: Proceedings of the IFIP 10th World Computer Congress, pages 1069-1076; North-Holland, 1986.

[3] Pierre-Arnoul de Marneffe; Holon Programming: A survey; Université de Liège, Service Informatique, 1973. Quoted in: Donald E Knuth;

Structured Programming with **go to** Statements; ACM Computing Surveys Volume 6 Number 4, pages 261-301, December 1974.

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

[5] David Garlan and Mary Shaw. An Introduction to Software Architecture; in Advances in Software Engineering and Knowledge Engineering Volume 1, V Ambriola and G Tortora eds; World Scientific Publishing Co, New Jersey, 1993.

[6] Michael Jackson; System Development; Prentice-Hall International, 1983.

[7] Michael Jackson and Pamela Zave; Domain Descriptions; RE'93 Proceedings, pages 56-64; IEEE Computer Society Press, 1992.

[8] Ralph E Johnson; Documenting Frameworks using Patterns; OOPSLA'92 Proceedings, ACM SIGPLAN Notices Volume 27 Number 10, pages 63-76, October 1992.

[9] Ralph E Johnson; Why a Conference on Pattern Languages? ACM SE Notes, Volume 19 Number 1, pages 50-52, January 1994.

[10] Cliff B Jones; Systematic Software Development Using VDM; 2nd edition; Prentice-Hall International, 1990.

[11] Nancy G Leveson and Clark S Turner; An Investigation of the Therac-25 Accidents; IEEE Computer, Volume 26 Number 7, pages 18-41, July 1993.

[12] Bertrand Meyer; Object-Oriented Software Construction; Prentice-Hall International, 1988.

[13] Peter Naur and Brian Randell, eds; Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968; NATO Brussels, 1969.

[14] D L Parnas and J Madey; Functional Documentation for Computer Systems Engineering (Version 2); CRL Report 237, McMaster University, Hamilton Ontario, Canada, 1991.

[15] G A Polya; How To Solve It; Princeton University Press, 2nd Edition, 1957.

[16] Rubén Prieto-Díaz and Guillermo Arango; Domain Analysis and Software Systems Modeling; IEEE Computer Society Press, 1991.

[17] Charles Rich and Richard C Waters; Formalizing Reusable Software Components in the Programmer's Apprentice; In Software Reusability Volume II; Ted J Biggerstaff and Alan J Perlis eds; pages 313-343; Addison-Wesley, 1989.

[18] W P Stevens, G J Myers, and L L Constantine; Structured Design; IBM Systems Journal Volume 13 Number 2, pages 115-139, 1974; reprinted in Peter Freeman and Anthony I Wasserman; Tutorial on Software Design Techniques; IEEE Computer Society Press, 4th edition, 1983.

[19] Wladyslaw M Turski; And No Philosopher's Stone, Either; Information Processing 86: Proceedings of the IFIP 10th World Computer Congress, pages 1077-1080; North-Holland, 1986.

[20] J B Wordsworth; Software Development with Z; Addison-Wesley, 1992.

[21] Edward Yourdon; Modern Structured Analysis; Prentice-Hall International, 1989.

26/07/94
M A Jackson
101 Hamilton Terrace
London NW8 9QX
mj@doc.ic.ac.uk
jacksonma@attmail.att.com
+44 0171 286 1814 (voice)
+44 0171 266 2645 (fax)

file: sejrnl02.tex