Proc Convenció Informàtica Llatina, pages 47-51; Boixareu Editores 1985

The Youthful Immaturity of Software Engineering

Michael Jackson
Michael Jackson Systems Limited
London, United Kingdom

I would like to thank the Program Committee, and the President of the Program Committee, for inviting me to speak at your Convention. Membership of the computing community brings many benefits; one of the greatest of these benefits is the opportunity to travel all over the world and to meet people from many lands. It is a great pleasure to me to be here today, and I thank you.

My theme is the Youthful Immaturity of Software Engineering. One might think that Software Engineering, especially by the standards of an industry that did not exist 35 years ago, is now mature, well-established, and fully grown. The NATO Conferences on Software Engineering took place as long ago as 1968 and 1969, some sixteen years ago. We have seen the development of modular programming and structured programming, of program proving techniques and of object-oriented programming. We have seen the development of three major kinds of database technology — the hierarchical, the network, and the relational database. We have seen the rise of functional programming, and the conversion of John Backus, the father of Fortran, to an ardent adherent of the functional approach. We have seen the introduction of logic programming and expert systems and set-theoretic specification languages. We have Lisp and Lotus-1-2-3; we have Ada and Adabas; Focus and Ramis and RPGIII. Surely this rich catalogue of achievement is solid evidence of maturity and success? Surely as Software Engineers we can now take our rightful place among the chemical engineers, the aeronautical engineers, the electrical engineers, and all the other practitioners of established engineering disciplines?

Unfortunately, we can not. There are many brilliant people working in software, and there are many very successful systems. But neither the quality of the people nor the success of the systems rests on that commonly owned and understood culture of theoretical foundations and disciplined practice that characterises a mature engineering subject. There are many symptoms and causes of this state of affairs, and I would like to discuss some of them today.

One notable symptom is the attitude of many software development managers to university Computer Science courses. A few development managers place a high value on a Computer Science qualification, and recruit Computer Science graduates to work in their teams. But most do not. Some managers, especially in data processing, actually prefer to employ unqualified people: they believe that the Computer Science course positively harms the student, reducing his ability to work effectively in a software development team. It is not clear who is to blame. Perhaps the managers are simply foolish; perhaps they fear to employ people who know something they do not know themselves, who have qualified in a subject that was not offered twenty years ago when they themselves graduated from University. Perhaps the managers are right, and the Computer Science courses are very bad training for practical software developers: the courses teach set theory and formal grammar and recursive functions, while the practical developer needs to know how to write COBOL and how to use IMS or TIP. But however we apportion the blame, this indifference to a Computer Science degree distinguishes Software Engineering very sharply from the established disciplines. It is hard to imagine a manager in a construction company refusing to employ graduates in structural engineering, or a manager at Boeing or Lockheed refusing to employ qualified aeronautical engineers.

Another, related, symptom is that we specialise too little. We speak of Software Engineering as if it might be analogous to Chemical Engineering or Automobile Engineering; more properly, it is analogous to a combination of all the traditional engineering disciplines, to what we might call Physical Engineering. We do not expect to find a qualified Physical Engineer, capable of designing a bridge today, a chemical plant tomorrow, and an aeroplane the day after. The

traditional engineering disciplines are highly specialised by product, and we should expect Software Engineering to be the same. We should aim to train and deploy not Software Engineers, but Compiler Engineers, Spreadsheet Package Engineers, Operating System Engineers, and Engineers of Airline Reservation Systems. Some of these specialisations have emerged naturally, especially where software products are sold ready-made in competitive markets, and they furnish the best examples, we can offer of solid success in software creation.

One consequence of this comparative lack of specialisation is that we are too ready to seek a panacea, a medicine that will cure all diseases. Tools and techniques that are wonderfully effective for some aspects of the development of some systems are offered for universal use, for haling all aspects of all systems. Software development methods, specification languages, and programming languages are rarely offered with tightly restricted claims of usefulness. Some people tell us that Ada will make all other programming languages obsolete; that functional programming will replace procedural programming; that prototyping will replace the traditional cycle of specification, design, and implementation; that one day all programs will be proved correct before they are used. Of course, we do not take such claims too seriously: they are often made only implicitly, and we know that they reflect the exuberance of the enthusiast rather than a considered judgement. But they certainly also reflect an immaturity in our field, and betray some deep technical deficiencies.

Before exploring some of these deficiencies I would like to set the context by putting forward a view of certain kinds of software development. I have in mind particularly information systems and data processing systems, which form the major topics of this convention. These systems are concerned to compute about some subject matter that exists outside the system itself: the payroll system computes about the employees, their work, their holidays, their promotion, perhaps their families and their retirement; the sales order processing system computes about the customers, their orders, the goods they receive and the money they owe for those goods. I call this subject matter the 'real world' for the system. Usually, the customer for the software is directly concerned with this real world: it is the context of his daily work. He can obtain useful information from the system because the software, when completed, will embody a model of the real world: the software must maintain this model and use it to compute results that will be true of the modelled real world. To answer the question 'how many employees are now on holiday?' the software examines its internal model employees: the answer will be true of the real world if the model is properly constructed and maintained. The software development task therefore involves at least these activities:

- making an abstract description of the real world, realising a software model according to the description,
- realising the mechanisms necessary to derive the required information from the model, and
- putting the realisations in a form capable of efficient execution by the computing facilities to be used.

The abstract description of the real world is the most important part of the specification: it is the medium by which the developer and his customer capture and record their common understanding of the problem domain, of the subject matter of the software. It also captures, in a general way, the functionality of the system: the system can produce any information that can be derived from the model realised according to the description.

Taking this view of the development task, we can immediately see two difficulties: one is a difficulty of language, and the other a difficulty of structure. The difficulty of language is that the language appropriate to the description of the real world will not, in general, be appropriate to the statement of an efficiently executable form of the system: that is, the specification language will not also be the implementation language. The difficulty of structure is that the structure of the real world description will not, in general, be the same as the structure of the executable system: that is, the structure of the specification will not also be the structure of the implementation. These are obvious difficulties, and have been recognised for a long time; but

they have not yet, I suggest, been satisfactorily solved. The technical deficiencies I referred to earlier are essentially the absence of satisfactory solutions to these difficulties.

Let us consider the language difficulty first. An old way, now discredited, of dealing with it was to write the specification in a language appropriate to the implementation. Customers for some data processing systems were confronted with a specification written in terms of files and programs, record layouts and logic flowcharts; they were invited, or even compelled, to 'sign off' such a specification before the developers would begin the implementation work of producing the COBOL and JCL texts and instructions to the machine operators. Inevitably, the customer was unable to understand such a specification, and disaster often followed: the resulting system was not well fitted to the customer's needs, and in more than one case was discarded after much time and money had been spent. The unhappy customers often complained that the data processing people were very clever but did not 'understand the business'. I would rephrase the complaint: the data processing people did not share the customer's view of the real world about which the system was to compute.

To achieve the necessary shared view of the real world the developer must describe that real world in terms immediately and perfectly intelligible to the customer. That means that the description must be the customer's own description, formalised only far enough to eliminate ambiguity and to ensure that there is no internal contradiction. It is not satisfactory to describe the real world in alien terms, showing mathematically that the alien description is mathematically equivalent to some other, unwritten description: the customer is rarely a mathematician, and is unlikely to be able to understand the equivalence well enough to agree confidently to the alien description.

An important example of this difficulty is provided by the common practice of describing the real world in terms of a database schema, or some syntactic sugaring of a schema. Undoubtedly there are some real worlds that can be adequately described in this way, worlds in which there is no time dimension and no events: only an unchanging set of objects with unchanging attributes and unchanging relationships. But for most information systems the time dimension in the real world is of central importance. I invite you to try the experiment of asking a layman, preferably one untainted by previous experience as a data processing customer, to describe the context of his work: he will most likely tell you something about what happens, and then what happens next, rather than something about entities and their attributes and relationships. He will not say: "there are suppliers, and for each supplier a set of products available from that supplier"; he will say rather: "we order products from our suppliers; they deliver the product to our warehouse, and invoice us for the cost; sometimes they deliver according to a schedule which we have agreed with them previously". For such a customer, the natural and appropriate language of specification is not a language of entities and attributes and relationships; it is a language of events occurring in a time dimension. But we are inclined to write a database-oriented specification, because we know that the eventual implementation will be in terms of a database.

Many specification languages have been proposed, each dealing well with some aspects of some real worlds. For example, Hoare's CSP can be used to describe time-ordered event sequences. To handle the simplest form of the difficulty of language, we need at least to be able to transform from one language into another: for example, to be able to transform from a specification of event sequences into a database implementation. But the difficulty does not end there. A significant real world is likely to have many important aspects, and one language will not be suitable for the whole of the specification. Some parts of the specification will need to be written in terms of event sequences, while others are best expressed in terms of entities and relationships, and others again will need to be expressed in a functional notation, and yet others perhaps in the language of sets. We are sadly lacking in the means to compose specifications expressed in different languages.

The structure difficulty is no less serious. One bad effect of the top-down, stepwise-refinement culture is felt here: there is a clear implication that progression from specification to implementation is by detailing or refinement — in other words, that the structure of the implementation is the same as the structure of the specification, only more detailed and having

machine-oriented lower levels. This is a serious mistake. There is no reason to suppose that the structure of the real world, as properly described, will mirror the structure of a convenient and efficient implementation: on the contrary, there is good reason to expect the opposite. Consider those aspects of a payroll system that are well captured by a specification of events within a time dimension. In the real world there is, roughly, one sequential process for each employee, and all these processes are progressing concurrently. The structure, therefore, is of very many processes, each taking tens of years to execute. In computer systems available today, such a configuration of processes can not be executed directly: arrangements must be made to reconfigure the processes so that their execution can be distributed over a small set of batch and on-line programs referring to a database. This reconfiguration requires the use of some transformation mechanism, able to transform not only the detailed content of the specification but also its gross structure.

In some systems, especially those where execution efficiency must be very high, there will be a need for very powerful transformations for another reason also. This reason is the need to apply the engineering principle known as the Shanley principle, first mentioned by de Marneffe in the context of software engineering. An example of the application of the Shanley principle is found in space rockets. A top-down designer might recognise that a rocket must have three parts (among others): an aerodynamic shell, a strong frame, and a container for liquid fuel. Applying the Shanley principle, the engineer implements all these parts in one: the outer shell of the rocket has the required aerodynamic properties, it provides the required structural strength, and it functions also as the fuel container. Only by applying this principle can the engineer make the rocket efficient enough: without it, it would be too cumbersome and too heavy. In the same way, some software systems will require different parts of the specification to be satisfied by one part of the implementation.

In software engineering, we have one great advantage denied to the physical engineers. Our material is intellectual material, represented by texts in various languages; it is capable of direct manipulation by the computer, which can transform it from language to language, from one structure to another. Some considerable and successful work has already been done in program transformation. I believe that an enlargement of the scale and scope of work in that area can bring many benefits, and perhaps a dramatic increase in our capabilities as engineers.