Do we still need the system life cycle?

Technical Report No.135

Carol Britton

February 1992

Do we still need the system life cycle?

Carol Britton

School of Information Sciences, Hatfield Polytechnic
Herts AL10 9AB, England

Since the dawn of software systems the development process has been dominated by the traditional system life cycle. This is a route map, onto which a software project fits more or less comfortably, and which prescribes the stages through which such a project must pass on its way from being merely an idea in the customer's mind to implementation and delivery. It is true that, in recent years, techniques, such as prototyping and the use of formal notations, have been developed which seem to encourage a move away from a rigid life cycle approach; yet software projects based on these techniques seem to be significant more for the fact that they do not follow a life cycle pattern rather than that they do introduce a new development framework.

In a typical life cycle model of system development one of the pricipal divisions is the demarcation between specification and design. The early stages in a development project aim to identify problems, establish the feasibility of pursuing a solution, analyse the current situation and determine what must be done to satisfy the customer's needs. These activities culminate in the production of a requirements specification which may be written using English, a formal notation, structured graphical techniques, or a combination of all of these. The main characteristic of such a specification is that it describes only *what t*the new system is going to do without any consideration of *how* this is going to be achieved. As a system development project progresses into the design stage of the life cycle the area of concern is widened to include such issues as the choice of appropriate hardware and software, design of the user interface and allocation of storage. The software designer uses the given specification as a starting point for decisions as to *how* the new system is going to implement the defined solution.

The split between specification and design, the *what* and the *how*, has always been one of the principal tenets of the traditional life cycle. It is argued that, only by abstracting away from details of implementation in the early stages of development, can the software developer hope to establish exactly what is needed and so produce a system which is fit for the purpose.

It all sounds so plausible, so obvious; why then are we becoming more and more deeply embroiled in the now notorious 'software crisis'? Why is it that most systems still overrun their budgets, are delivered late and fail to do what the customer wanted? Is the life cycle - and therefore the way we develop most of our systems - fundamentally flawed?

It is true that the structure of our traditional systems life cycle may no longer be appropriate as a framework for modern systems development. We need only to think of the many tools and techniques that have been introduced since the advent of the life cycle: fourth generation languages, prototyping, formal specification languages and automated project management tools are but a few of these.

The most significant change, however, has not been in development process itself, but in the type of system required. As recently as a decade ago much of the work of a software systems developer involved building complete systems for customers who had little or no experience of computers. Today virtually everyone has used or come into contact with a computer system. It is increasingly rare for a developer to be asked

to build an automated system to replace a wholly manual one. More and more systems are 'second time round' and all the easy jobs have been done. The old life cycle view of developing a system from the first vague thoughts right through to implementation and handing over to the customer is no longer appropriate. It is about as unrealistic as the idea of building a new road and being able to make a free choice about where it is to go and what route it will take. Our country is full of roads and our computers are full of software. Unless we start building motorways under the sea any 'new' road is merely a modification of the existing network. In the same way, believing that the job of a software developer is to build new systems, is simply self delusion. With computer systems already in place, however inefficient they may be, the system developer's tasks today are those of extension and modification.

But does this actually have any bearing on the way we develop systems? The traditional life cycle explicitly takes account of the need for change in any system. In nearly all life cycle models there is a 'Maintenance and Modification' stage which encompasses everything from fixing minor bugs in the code to major alterations in the system functionality. Most system development methodologies state euphemistically that even small changes should involve working through all the development stages.

To my mind, this is simply patching over the problem. The fact that a computer system is already operational gives rise to constraints on the development of a new system which are of a different order from those imposed by an existing manual system. Time, money and effort will already have been invested in the original computer system and organisational procedures based on it will have been developed.After the major upheaval of installing a computer system first time round, very few customers are going to be prepared to jettison what is already in place in favour of a totally new system. It is essential, therefore, that the existing computer system is analysed in depth, that the constraints it imposes on future development are identified and that these constraints are borne in mind throughout the development of the new system.If we take this view of modern system development, we can see that the life cycle in its present form is no longer appropriate for the type of work that system developers are asked to carry out since it is based on two false premises: first that the system is to be developed from scratch, and second that what must be done can be decided without detailed consideration of how this is to be achieved.

It is true that certain methodologies, such as SSADM, begin the development process by modelling the current system. In the case of SSADM this is done principally by means of current physical data flow diagrams which aim to capture *how* the system (manual or automated) functions at present. These data flow diagrams do indeed give the developer a picture of the current system implementation and could therefore be used to ascertain how this implementation will affect future development. In practice, however, the current physical model of the system is used as a basis for abstraction to the current logical model in which the details of implementation are not considered relevant.

What is needed is an explicit stage in the life cycle during which the model of the current physical system is analysed in depth to determine precisely what the starting point is for the new system. It is at this point that issues should be discussed such as the choice of hardware and software, the design of the user interface and measures of performance. The customer's requirements, while perfectly reasonable in themselves, may be totally impractical as an extension to an existing system. There is little point in a developer spending time and effort defining *what* the new system is to do if this can only be achieved through hardware or software which is not compatible with that of the customer's current system. Many of the major decisions about the new system will be dictated by the nature of the existing system in the customer environment and the degree to which the client is prepared to adapt both the system and

the ways of working which have developed round it. It is essential for the success of 'second time round' systems that an in-depth analysis of the current system implementation is carried out and the resulting constraints on the new system are identified at the very beginning of the development process.

The fact that most system development projects today replace or modify existing computer systems also has implications for the relationship between the customer and the system developer. Not many years ago most customers were novices who were just moving into the world of computerization. They had faith in the new technology, were eager for change and felt confident that installing a computer would be the solution to their problems. Most customers had little or no knowledge of computers and so tended to regard the system developer as a technical genius who could provide the perfect system with minimal user involvement. This view of the customer/ developer relationship is supported by the traditional system life cycle and methodologies based on it which explicitly involve the customer only at the very beginning and end of the development process.

Today few customers believe that the developer can wave a magic wand or that the computer can solve all their problems. Customers now have experience of a computer in the problem environment and rightly believe that their expertise is as valuable as that of the developer. The former customer/expert relationship is now much more a partnership. This view is not reflected in the life cycle approach to systems development, but is found in techniques such as prototyping where the role of the customer in designing the system is just as important as that of the developer.

Does this mean that the life cycle is no longer of use to system developers? It seems that the traditional separation between the *what* and the *how* is no longer applicable to most present day development projects, nor does the life cycle framework cater for the present day relationship between customer and developer. Have we reached the point where it is old-fashioned and misguided to base system development projects on the life cycle?

However tempting it may appear, it would be a disasterous step simply to dispense with the life cycle. The framework and management support that it offers are too valuable to throw away while we have nothing with which to replace them. The factors that originally gave rise to the life cycle are still with us. Problems such as the complexity and scale of systems required, the size of development teams and the speed of change in the computing industry are, if anything, more accute today. The life cycle can go some way towards alleviating the situation by imposing a structure on the development process, identifying milestones, encouraging standardised documentation and acting as a basis for project management.

In the current state of software system development it is relatively simple to criticise the life cycle and to pinpoint some of its underlying problems. It is very much harder to suggest how these problems might be solved. This paper identifies two distinct areas which could make the life cycle a more appropriate basis on which to build today's software systems. First there should be a comprehensive study of the existing system, including current hardware, software and organisational procedures. At this stage it must be established which of these the customer is prepared to change and which are to remain in the new system. This information, together with the customer's problems and requirements, should then form the basis of all decisions about the new system.

The second area of the life cycle which could benefit from change is more difficult to define, since it involves a view of the customer / developer relationship, rather than a particular development activity. The customer is an essential part of any system development process and should not be pushed aside at certain stages because these are seen as of concern only to the system developer. If the customer were having an

extension built to his home he would not presume to understand all the architectural and engineering details of the process, but he would expect to be able to ask at any stage why things were done in a particular way and to understand the answers he was given. In building software systems the developer is answerable to the customer in just the same way an architect or a builder. At any stage the developer should be prepared to explain to the customer not only *what* and *how* but also *why* development is proceeding in a particular way. If this degree of customer involvement disrupts the management structure provided by the life cycle it may be frustrating, but it is vastly preferable to a dissatisfied customer and a system that falls short of expectations.

Whether we need some form of life cycle, guidelines, framework, or whatever we choose to call it, is not the question. What we need to ensure is that our structure is rigid enough to provide support for managers and developers of systems, while at the same time flexible enough to respond to computing's rapidly changing environment. For many years the system life cycle has been a life line for system developers; now we must make sure that it does not become a noose.