TELEMETRY SOFTWARE DEVELOPMENT LIFE CYCLE

Dr. Alan B. Campbell
President
ABC Systems, Inc.
3536 Millikin Ave., San Diego, CA 92122

ABSTRACT

All orderly software development proceeds through the phases of a predictable life cycle. This behavior is characteristic of telemetry software development, also. Each phase of the life cycle is definable in terms of specific milestones. Understanding the life cycle is crucial for accurate estimation of time and effort, as well as for producing reliable software on time and within budget.

INTRODUCTION

The concept of the software life cycle is a simple, powerful, unifying idea in software engineering. The fundamental principle behind the concept of the software life cycle is that orderly software development proceeds through predictable phases. For example, three phases through which software development always proceeds are definition, implementation, and maintenance. Each of these phases in turn can be broken down into identifiable subphases. Each phase and subphase must be defined so that it is delimited by the achievement of concrete milestones. As an example, the definition phase is ended by the production of the requirements and specification documents for the software, which allows the beginning of the implementation phase. In general, a subsequent phase must not begin until the previous phase has been completed, as evidenced by the production of the concrete end milestones. These hard and fast limits are seldom achievable in practice, but a realization of the manner in which phases should follow one another will make clear the hazards that can occur when an orderly plan is not followed. Practice has shown that the vast majority of software development difficulties have come about through beginning development phases too early; the most common and devastating such error is to begin coding before the requirements are truly defined.

Many difficulties in the production of software can be traced to a lack of understanding of the constraints imposed by the software life cycle. An understanding of this cycle not only is necessary for the development of high quality software in a reasonable time, but is also required for an accurate estimate of time and resources for the development effort. The

characteristics of the software life cycle are especially important for telemetry software, because telemetry software is often under development at the same time as both the telemetry hardware and the system to be tested. Because of this process of simultaneous development, changes in the other systems will often occur that force the telemetry software development backwards into earlier phases. Any effective attempts to deal with the consequences of such changes on the telemetry software development will require an understanding of the software life cycle.

The purpose of this paper is to examine the elements of the software life cycle, relate that life cycle to typical telemetry development, and encourage further investigation and understanding by those who are involved in the development of software for telemetry.

SHAPE AND PHASES OF THE SOFTWARE LIFE CYCLE

The general shape and phases of the software life cycle are indicated in Figure 1. Certainly the overall shape squares with normal experience-- the effort is low at the beginning, reaches a peak near the delivery time, and then tapers off to a long-term low level of effort for continued maintenance. Putnam (1) has argued persuasively that the shape of the curve after the specification phase is that of the Raleigh distribution, in which programming effort is of the form $2Kat[\exp(-at^2)]$. Lehman (2) has cast doubt on the use of the Rayleigh model for purposes of estimation, but Putnam's work is certainly illuminating inasmuch as his model clarifies indisputable relationships between the development time, the state of technology applied to the problem, and the total effort required for the development. Putnam's work even provides a quantification for Brooks' law ("Adding manpower to a late software project makes it later."[3]). If we assume that his Rayleigh distribution is a reasonable representation for the effort of a project, then the product of effort, K, and the fourth power of the time to develop, t_d^4, is a constant. Thus in order to halve t_d we must provide 16 times the manpower. And a further virtue of Putnam's work is that he shows quantitatively that even with unlimited manpower there are limits as to how rapidly a job can be done.

Figure 1 shows the succession of phases of which the software life cycle is composed. Actually, of course, the transitions between phases are not so distinct as the drawing indicates. Software development is a process of successive refinement because as each new phase is entered facts will often emerge that will necessitate some improvements of the work performed in previous phases. However, this fact must not be allowed to prevent the production of the milestone documents that mark the dividing lines between phases. Also, milestones must be concrete: "Requirements document signed off by Project Manager," NOT "Requirements document 95% complete." The lesson to be learned from a realization that software development involves successive refinements is that the milestone documents must be as easy as possible to revise in a disciplined manner, and no

one should resist the inevitable necessity of changing requirements as the project progresses. “Freezing” a design is almost always an illusion during development, and those involved must simply accommodate their procedures to that fact.

The particular phases that are indicated on the life cycle diagram are to some extent arbitrary. The normal succession of activities has been broken into phases in many different ways, from the four phases into which Mitre has decomposed the process (Conceptual, Requirements, Development, and Operations) to the eight phases into which TRW has decomposed the process. The particular decomposition is not important (it is easy to relate the different particular phases to each other), but it is important to recognize that software development does progress through phases, with each phase having its own characteristics and requirements.

The phases that usually get too little attention are those before the implementation begins. It is difficult to overemphasize the necessity of good initial planning and specification before the coding begins. All too commonly, the programmers start coding before the problem has really been defined, with predictably bad results. It sometimes seems that there is a conspiracy between management and the programmers to guarantee this too-early start: the programmers want to get in there and start coding right away, and the managers are made nervous if they don’t begin to see lines of code (“Why aren’t they programming like they’re paid to do?”). This almost universal tendency makes it all the more important that everyone involved understand the life cycle and insist that the appropriate milestones have been reached before beginning another phase.

The software life cycle phases and the milestones that indicate their completion are given below.

Phase	Milestone
1. Study Phase,	Feasibility Document
2. Requirements Phase,	Requirements Document
3. Specification Phase,	Specification, System Test Plan
4. Estimation Phase,	Budget and Schedule
5. High Level Design Phase,	Software Design Document
6. Detailed Design Phase,	Module Designs
7. Programming, Test, Integration Phase	Debugged Programs
8. System Test Phase	Test Results
9. Maintenance Phase	

Note the number of phases that are not the actual production of programs. Especially note that six phases occur before coding ever commences. It is a common mistake to think of the production of software as just the coding activity. Brooks has said (3) that a properly-scheduled software task should provide for 1/3 of the time and effort for planning, 1/6 for

coding, 1/4 for component test and early system test, and 4 for final system test. Brooks emphasizes that the initial planning activities are crucial, and also that 1/2 of the time will be spent on testing, whether that much time has been scheduled or not-- and because most schedules do not allow that much time, this is a typical reason for slipped schedules and missed delivery dates.

Note also that the System Test Plan is produced with the Specification. The Specification should be complete and definite enough so that all system testing is performed to insure agreement between the Specification and the completed software.

In the Estimation Phase, accurate estimates of the software effort and time are probably impossible without a reasonable understanding of the realities of the software life cycle. The ideas advanced by Putnam (1) should certainly be studied by anyone involved in software estimation.

The two phases that deal with software design will take the overall software concepts and supply all the details of software organization and interrelation. The High Level Design Phase will be concerned with the overall number of modules, their interrelations and interconnections, and the commands, flags, and data that are passed between them. The Detailed Design Phase will produce those elements from which the actual coding is done. Some examples of design elements are truth tables, structured English, and flow charts. There are books currently available that deal with what is called the structured design of software (4,5).

The phase that contains the actual programming is identified as the Programming, Test, and Integration Phase. If the software production is properly top-down, these three activities will take place in a closely interrelated manner. The top-most module will be written and tested with the lower-level modules stubbed off. Then a second-level module will be written, individually tested, and integrated with the top module. Proceeding in this manner has several advantages. First, the top-most module, which is probably the most important, gets the most use and is thus the best tested. Second, each module is made to operate by itself before being integrated with other modules. Third, integration takes place by easy stages, which helps prevent the chaos that sometimes occurs when large systems of modules are integrated. One useful modification to this strict top-down implementation involves simultaneous early production of the lowest modules. It is good to write these modules early because they usually deal with interactions with people, which must be defined at the outset. With this method integration proceeds from the top and bottom at the same time.

REPEATING PHASES

As was mentioned earlier, software production seldom progresses smoothly from phase to phase. Portions of phases will have to be repeated as the project evolves and problems are better understood. As Brooks has said (3), “For the human makers of things, the incompletenesses and inconsistencies of our ideas become clear only during implementation. Sometimes the basic requirements themselves will change as the customer comes to understand his problems better. Those who are involved in the tasks of providing software must be aware of this necessity to repeat or modify earlier work so that they can plan to be able to accommodate such changes. One thing that must be kept in mind, however, is that such repetition will have an effect on the schedule. If the estimates that produced the original schedule were good estimates when they were made, then repeating a phase should not lessen the time required for later phases. What this means is that a delay in an early phase must be assumed to cause a comparable delay in the end date. There is a tendency for people to think they can make up time lost in the beginning by doing subsequent phases more quickly than the original estimate.

Making up lost time is sometimes possible, but only through the application of enough more resources truly to make a difference. Wishful thinking will not make the production of software go more quickly. Gordon and Lamb (6) have argued that Brooks’ law may not always be so hard and fast as Brooks has said. Their statement: “Adding people to late software projects sometimes helps, but only if more people are added than expected to be needed, and only if they are added sooner than they are expected to be needed.

TELEMETRY SOFTWARE

A great deal of telemetry work is done in development, during which time the conceptions of the project are often changing. Thus more than many other software efforts, telemetry software must be so designed as to facilitate the inevitable changes. The sources of these changes are many. Often some of the fundamental ideas of the experimenters change. In many instances the characteristics of the end instruments are different from what was expected, or the end instruments must be changed to meet the changed experimental concepts. Not only is the system to be tested changing to some extent, so also is the telemetry system itself. As the telemetry system development proceeds, the concept of that system too changes.

What steps can the telemetry software developer take to facilitate the inevitable changes? The steps he should take are just those of typical good software design and implementation. First, the software should be highly modular, so that it becomes relatively easy to add, delete, or modify individual software functions. Also, relations between modules must be well-defined. The software implementer must avoid building constants

into his programs. What appears to be a constant (such as the number of millivolts per bit) should be treated as a variable and defined only in one place. Thus if that value must change, it needs to be changed only in that one place. A methodical use of this technique will allow the software to remain light on its feet, so that it can respond most easily to changes in fundamental numbers. The use of tables is important, too, for the same reasons. There is one other essential for modifyable software: record-keeping of the different software versions for each module must be scrupulous, and must be available to every worker. It is an unforgiveable waste of resources for someone to integrate his software with other software that he did not know was out of date. There are excellent books available on the subject of good programming practice (e.g., 7).

Thus telemetry software should display two main general characteristics. First, the code should be constructed using the now well-known rules for good software. Second, even more than most code, telemetry software should be designed to facilitate those changes that must occur during the system development. In terms of the software life cycle, the phases that occur before the coding are even more important than with some other types of software. The reason for the extra importance of these early phases is that planning to accommodate change must especially be addressed during the requirements and specification phases, and also especially during the design phases.

CONCLUSIONS

The overall behavior of the software development life cycle is well-known and reasonably predictable. Efficient production of software will take into account this life cycle and will both accommodate and use it. The transition from one phase to another must not occur until the appropriate, definite, measurable milestones have occurred (not 95% complete”). The early definition phases have been discovered to be the most important, because the production of something as complicated as software requires excellent initial planning. The early phases are even more important in the production of telemetry software than for other software, because telemetry software must be planned so as to be exceptionally able to accommodate change.

FINAL OBSERVATION

The production of software has been well-studied in the last several years, and many aspects of it are nor reasonably well understood. Anyone who is involved in the production of software who does not acquaint himself with the existing literature on software production and utilize the lessons that have been so painfully learned is remiss, and is certainly culpable when the almost-inevitable software disaster occurs.

REFERENCES

1. Putnam, Lawrence H., “A General Empirical Solution to the Macro Software Sizing and Estimating Problem,” Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers. The Institute of Electrical and Electronics Engineers, New York, N. Y., 1980, pp. 141-157.

2. Lehman, Meir M., “Programs, Life Cycles, and Laws of Software Evolution,” Proceedings of the IEEE, Vol. 67, No. 9, pp. 1060-1076, September, 1980.

3. Brooks, F. P. Jr., The Mythical Man-Month-- Essays on Software Engineering. Addison-Wesley, Reading, MA., 1975.

4. Page-Jones, Meilir, The Practical Guide to Structured Systems Design, Yourdon Press, New York, N. Y, 1980

5. Yourdon, Ed., and Constantine, Larry L., Structured Design, 2nd Edition, Yourdon Press, New York, N. Y., 1978.

6. Gordon, R. L., and Lamb, J. C., “A Close Look at Brooks’ Law,” Datamation, June, 1977, pp. 81-86.

7. Kernighan, Brian W., and Plauger, P. J., The Elements of Programming Style, 2nd Edition, McGraw-Hill, New York, N. Y., 1978.

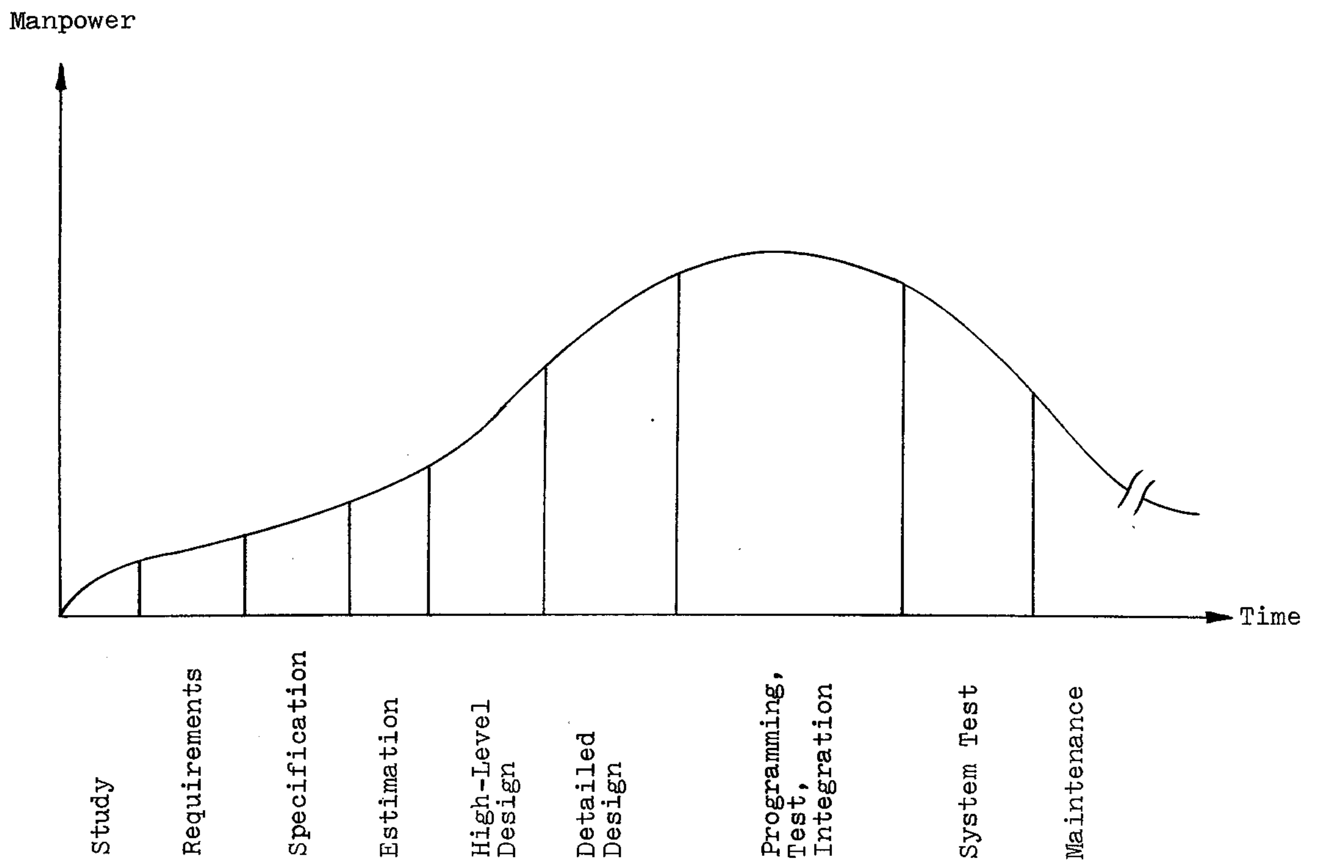

Figure 1. The Shape and Phases of the Software Life Cycle