What life? What cycle?

by NICHOLAS ZVEGINTZOV
Staten Island, New York

ABSTRACT

The traditional system life cycle model does not portray the life of a system, nor is it a cycle. An alternative model is described that portrays the modification cycle of the system and the detailed activities of making a change. Implications are drawn for maintenance, development, and the education of software engineers.
THE NOT-LIFE NOT-CYCLE

During the 1970s the phrase *system life cycle* came to be widely used to describe the stages of growth of an applications system. For a while the phrase was almost a fad or a buzz word. As Glass and Noiseux say in their *Software Maintenance Guidebook*: ²

At a recent computing conference, discussion of the so-called computing life cycle became a standing joke. Every presenter of every paper showed a viewfoil or a slide containing his or her graphic version of the concept. Toward the end of the day, one wag referred to his as the "obligatory software-life-cycle chart"!

These charts had the basic form of Figure 1, in which a system comes into being by being elaborated or made concrete in a sequence of phases and finally is installed and enters an operation and maintenance phase. The exact number and names of the phases sometimes varied, but the basic structure remained the same. The arrows were generally downward to indicate the management constraint that each phase must be frozen before the next is started, although sometimes feedback arrows were added to indicate that this was not always possible to enforce (Figure 2).

A Finnish version even offers a more elaborate pattern of feedback (Figure 3).

Popular though this model is, there are two objections to calling it a life cycle:

1. It does not portray the system’s life.
2. It is not a cycle.

First, it portrays only the creation, development, or youth of a system, and does not include its adulthood—the productive phase of its life. It is as vague about the operation and maintenance phase as a teenager is about life after marriage.

Second, it is a linear path or progression from goals to operation and maintenance; and it does not, as a cycle must, in some sense return to its own beginning. In borrowing the term *life cycle* from biology, the originators of this model failed to borrow its central concept, the tracing of the organism from its embryonic origin to the adult state in which it originates and nurtures the embryo of a new individual.

---

**Figure 1—"Life cycle"**

**Figure 2—"Life cycle" with feedback**
Goals
Requirements
Design
Code
Install
Operation and Maintenance

Figure 3—Finnish “life cycle”

Thus this model is misnamed if it is regarded as a model of the system life cycle. It is, in fact, a model of the development path of a system—in fact, as I shall show in this paper, of part of the development path. Nevertheless, it has had great popularity, since, even misnamed and partial as it is, it helps to illuminate and analyze (and hence make controllable) a significant and expensive portion of the system’s life. Can we supplement it by finding an equally illuminating model of the rest of the life of the system?

THE MODIFICATION CYCLE

The first step is to use the clue given in the name of the last phase of the development model—operation and maintenance. There are two terms here, representing two activities. The system does its job (operation), and it undergoes modification (maintenance). A relatively small part of this modification consists of repair; the rest consists of changed or enhanced function. (Readers who doubt this about software systems are referred to the Lientz and Swanson and U.S. General Accounting Office questionnaire studies.47)

The general categories of such changes, for organizations as well as for computer systems, are as follows:

1. New, changed, or deleted functions
2. Adaptation to environmental changes (legal, financial, political)
3. Consolidation, reorganization, routinization
4. Turnover of staff, equipment, resources

A simple model of the cyclic incorporation of such changes is portrayed in Figure 4. During the system’s operation, its constituents (owners, managers, operators, users) assess its performance. On the basis of this assessment, they generate requests for change related to their own interests. These, after political tradeoffs and the commitment of resources, become modifications of the system. These modifications affect the system’s performance, which the constituents assess, and the cycle continues.

This model is cyclic and does deal with the adult life of the system. The developmental model can be grafted onto it to form what I call the starting gate model, because development appears as a one-time initiation, after which the system continues infinitely around the cyclic track (Figure 5).

This model, though better than the last, is still inadequate as a portrayal of the life cycle. In particular, the cyclic part of the model appears impoverished and undifferentiated compared to the richness of structure shown on the developmental starting gate. Have the developmental stages of goals, requirements, design, and code anything to tell us about the modification of existing systems? Certainly. Although they do not appear as developmental stages, they are the analytic framework we need for comprehending (and therefore controlling) an existing system. With this hint, we can proceed to fill in the detail of the modification cycle.

LEVELS OF DESCRIPTION

Why is development performed in stages? Simply because the gap between goals and code is too great to be crossed in a single intellectual leap. Therefore the process is converted into a chain of stages chosen so that the gap between each stage and its successor can be crossed.

Clearly, the same problem exists in understanding existing systems. The gap between high-level performance and low-level implementation is too wide to be crossed in one leap. The leaps must be narrowed by interposing various intermediate layers to aid understanding; these are the levels of description.
The levels of description are alternative and simultaneously correct descriptions of an existing system from different perspectives. In general, a higher-level description answers the "Why?" question for a lower, a lower answers the "How?" question for a higher, and the levels of description roughly correspond to levels of management. Each level has a characteristic vocabulary appropriate to the people who deal at that level, and each level contains motivating information not directly derivable from any other level. Thus the levels contain information-hiding decisions in the sense used by Parnas,5,6 and they correspond to the "knowledge domains" found by Ruven Brooks in documentation.1

In Figure 6 the levels of description are shown under an existing system; the stages of development are shown under a desired system; and labels are given to equivalent levels, stages, and management roles. (This particular example is of a system of a scale large enough to match the major operations of the organization, and therefore its goals reach as high as the chief executive officer; but there are many systems with more modest goals reaching less high in the hierarchy.)

1. PERFORMANCE/GOALS/CEO: "We (need to) have an inventory control system for our warehouses and distributors."  
2. CAPABILITIES/REQUIREMENTS/V-P: "I (need to) control the regional warehouses at . . . ."

3. FUNCTIONS/DESIGN/MIDDLE MANAGEMENT: "I (need to) have staff, procedures, and equipment to handle picking, shipping, reordering, charging . . . ."  
4. CODE/CODE/LINE MANAGEMENT: "I (need to) have terminals and display/update software for my forklift operators . . . ."

On both the Existing and the Desired sides the levels (stages) describe (define) the same system, but there are two differences in the way that they relate to each other.

First, the levels of description exist simultaneously—they represent the state of the system as viewed today at different levels of management. By contrast, a desired system goes through the stages of development in sequence. This difference contrasts the actuality of an existing system with the futurity of a desired system.

Second, the identity of the levels of description is imposed from below, whereas the identity of the stages of development is imposed from above. The managers of an existing system must understand it, warts and all. If a higher-level description does not accurately abstract a lower-level one, it must be revised. By contrast, a desired system is described in terms of intention. If a later stage does not accurately implement an earlier one, it is reworked until it does. This is why the arrows are upward on the Existing side and downward on the Desired side.

The levels of description have brought into the analysis of an existing system the richness of detail found in the development model. How do these levels interact with the change process to give a model of the modification process?

HOW TO MAKE A CHANGE

The stages of making a change are as follows:

1. Understand the request.  
2. Transform the request to a change; the change is the goal.  
3. Specify the change: choose cut-line and patch.  
4. Develop the patch.  
5. Test.  
6. Install.

The first stage in making a change is to understand the request. A request is a description of a new system, phrased in
terms of the existing system and in the vocabulary of a level of description:
"Computerize inventory!"
or
"Get ready for the new Denver warehouse."
or
"Put I/O devices on the forklifts."

Understanding a request (Figure 7) requires (a) understanding the system via its levels of description, and (b) running the request down the levels of description until it finds a place where it makes a difference.

EXISTING SYSTEM

Performance
Capabilities
Functions
Code

Figure 7—Understand the request

One may assume that the request "Computerize inventory!" affects the system even at the level at which the CEO views it. However, the request "Get ready for the new Denver warehouse" would cause only minor changes in a high-level document that observes that "All regional warehouses are computerized.... " Going lower still, the request "Put I/O devices on the forklifts" does not affect the CEO or the V-Ps, but only the regional managers and below. Thus the process of understanding a request is a process of working down the levels of description to find the highest level affected.

TRANSFORM THE REQUEST TO A CHANGE

The second stage in making a change is to transform the request to a change (Figure 8). The action here is to apply the request to the description of the existing system and derive an alternative description of a desired system, i.e.:

1. Given the existing inventory system, what would a computerized one be?
2. Given the existing computerized system, what would one be that includes the new Denver warehouse?
3. Given the existing in-warehouse system, what would one be that includes I/O devices on the forklifts?

At the point at which the request becomes a change, the system begins to bifurcate: above it there is no distinction between existing and desired, below it there is one. Closing this bifurcation is the goal of the change process. (Note that this goal is usually at a level much lower than the goals of the overall system.)

SPECIFY THE CHANGE: CUT-LINE AND PATCH

The third stage in making a change is to specify the change; this involves choosing the cut-line and the patch (Figure 9). The cut-line is existing code or procedures which must be changed; the patch is new code or procedures. In the following simple example, the cut-line is the sentence from IF to PERFORM SECTION-ELSE, and the patch is SECTION-Y.

EXISTING CODE

IF INPUT="X"
THEN PERFORM SECTION-X
ELSE PERFORM SECTION-ELSE.

SECTION-X. (TEXT)
SECTION-ELSE. (TEXT)

CHANGED CODE

IF INPUT="X"
THEN PERFORM SECTION-X
ELSE IF INPUT="Y"
THEN PERFORM SECTION-Y
ELSE PERFORM SECTION-ELSE.

SECTION-X. (TEXT)
SECTION-Y. (TEXT)
SECTION-ELSE. (TEXT)

Choosing the cut-line is the major intellectual challenge in making a change. The cut-line has less code than the patch, but it has greater complexity, since it derives complexity from its intimate interaction with the complexity of the main system. The designer has two aims in choosing the cut-line:

1. Minimize the impact of the cut-line on the existing system.
2. Isolate the patch from sources of variability in the existing system.

It is important to minimize the impact of the cut-line, because it directly affects the fabric of the existing system. In changing a system, as in surgery, the major challenge is not causing the desired alteration in the organism, but avoiding undesired alterations.
It is important to isolate the patch from sources of variability in the existing system because the patch is a piece of free-standing code, possibly of considerable size, and its development may be a development project of considerable scope. To give this project the best chance of succeeding, it is desirable to isolate it as far as possible from other ongoing or incidental changes of the host code. Thus the correct choice of the cut-line is what enables the specifications of the patch to be frozen.

FINAL PHASES OF THE CHANGE

In fact, develop the patch is the fourth stage of making a change (Figure 10). This is where the classic life cycle, i.e. the development path, is an appropriate model. The project has a frozen goal: code to implement the new requested function. It also has frozen specifications, namely, to fit with the chosen cut-line. The design, code, and unit test of the patch therefore proceeds according to the development path model.

The fifth stage of making a change is to test it. This is done by inserting the patch at the cut-line and testing the new system in parallel with the old. The main tests are these:

1. Test for the enhanced functions of the patch ("Do we have what was requested?")
2. Test for degraded functions at the cut-line ("Have we lost what we had before?")
3. Regression test

The sixth and final stage of making a change is to install it—insert the tested patch at the tested cut-line and remove the old system and the entire scaffold of the change process. Then the changed system becomes the existing system, and the modification cycle begins anew (Figure 4).

The stages of the change cycle are portrayed together in Figure 11, which may be regarded as a detailed expansion of Figure 4. We propose that together they supply a more adequate model of the system life cycle.

CONCLUSIONS

The purpose of a model is to disassemble a complex process into its component parts so as to aid in the assignment of resources, the training of novices, the specialization of roles, and so on. The model given in this paper has already proved fruitful in explaining on a theoretical basis some of the things that maintenance managers do—for instance, the philosophy of the big patch, or the quick and dirty fix. But, beyond that, it has implications for development and for software engineering education.

In the model the development path from goals to installation—the entire classic life cycle as presented at the start of the paper—is seen as a model of patch development, which is just a part of the change cycle, which in turn is subordinate to the modification cycle of the entire system. A similar subordination is seen in Van Horn's recent model of "evolutionary software development," but it is not reflected in any of the curricula or textbooks for software engineers. Yet such a subordination has profound implications for the staffing, estimating, or teaching of software development.

Most development projects are not, in fact, the creation of something entirely new—they are creations of a replacement for a relatively small section of a relatively larger system. They are, in fact, patch developments. But, under this model, patch development entails also the analysis of the larger system, the transformation of a request into a change, the specification of the change via the choice of cut-line and patch, and a controlled turnover. In this context, patch development is rather a small part of the problem. In fact, the choice of the cut-line is seen as the major design challenge, since it provides the frozen goal that is the idealized prerequisite for the classic development model.

Yet patch development is the only model currently being taught to software engineers; the other aspects of system modification are only learned from on-the-job experience or apprenticeship. If the analysis in this paper is correct, it gives theoretical support to the view widely held by managers that software engineers do not come out of school well prepared for the realities of their job.

ACKNOWLEDGMENTS

I would like to thank my colleagues J. Cris Miller and Michael Lyons, and our students in the course Software Maintenance: Tools, Techniques, and Management Strategies, for pushing and pulling me through the painful process of clarifying this model.
REFERENCES

LANGUAGE AND DATABASE PROCESSING