Cognitive Approach to Systems Engineering and Re-Engineering: Integrating New Designs with Old Systems

JOSEPH M. SCANDURA
Structural Learning, Computer Systems, 3700 Walnut St., University of Pennsylvania, Philadelphia, PA 19104, U.S.A.

SUMMARY
Most MIS expense goes into the maintenance of existing software. Estimates vary from 50% to 60% at the low end up to 90%, even 95%, depending on responsibilities and the type of software. Coming up with better ways to maintain (i.e., understand, modify and update) existing software is unquestionably the most important challenge facing the software industry today.

Saddled with obsolete but essential software, a growing number of organizations are faced with a series of unpalatable choices. One option is to simply continue with the same old software, patching it where possible to meet the most pressing needs. This approach has a number of limitations not the least of which are lost opportunity and reduced competitiveness. Moreover, owing to unforeseen and increasing interactions, time per unit of change (especially structural change) tends to increase exponentially throughout the maintenance life cycle.

Eventually, old systems must either undergo a major overhaul or be replaced with new ones. Properly done, and implemented, either approach can add a measure of efficiency previously impossible—and cost recovery can often be accomplished over a reasonable period of time. The initial costs involved in such renewal, on the other hand, are often prohibitive.

Faced with this dilemma, what are decision makers to do? In this paper, I propose a new approach to system maintenance which can both dramatically improve systems and minimize renewal costs. In turn, I describe a typical maintenance scenario, and consider the use of CASE restructuring tools. Then I show how the human or cognitive factor impinges on the maintenance issue, and how this implies the need for abstraction. Next, we note limitations of standard ‘bubble chart’ and other notational schemes and introduce the FLOWform representation as a means of overcoming these limitations.

Finally, with this background, I describe the Cognitive Approach to Systems Engineering and Re-Engineering (CogApp). In this context, I summarize capabilities of the PRODOC reNuSys Workbench™ which has been used to implement the CogApp. I close with a short case history describing application of this approach, and the results obtained therefrom.

KEY WORDS

1 Results described in this paper draw on work using the PRODOC reNuSys Workbench™. More information on PRODOC™ can be obtained from Scandura Intelligent Systems, 1249 Greentree Lane, Narberth, PA 19072, U.S.A. (215-664-1207).

Received 11 November 1989
Revised 15 March 1990
TYPICAL MAINTENANCE SCENARIO

A typical maintenance scenario is as follows: Given a desired modification or improvement, the analyst or maintenance programmer first studies the existing code. Sometimes this includes looking at documentation or various cross-reference reports. The programmer spends most of his time, however, analysing the actual code. The documentation of old code, where it exists at all, is rarely current and cannot be relied upon in making changes.

Changes invariably are made in whatever way is easiest for the programmer in the short run. Supplementary conditions, for example, are typically added in one of two ways: (a) by supplementing old conditions with new ones or (b) by adding another level of structure to the code. For example, consider the following code:

\[
\begin{align*}
\text{IF } C &= c_1 \\
\text{THEN } A \\
\text{ELSE } B
\end{align*}
\]

Suppose we want to introduce a new condition relating to variable \( D \). Specifically, when \( C = c_1 \) we want to do \( A \) only if \( D \neq d_1 \). This can be done in either of the two ways shown below.

\[
\begin{align*}
\text{(a) IF } C &= c_1 \text{ and } D \neq d_1 \\
\text{THEN } A \\
\text{ELSE } B
\end{align*}
\]

\[
\begin{align*}
\text{(b) IF } C &= c_1 \\
\text{THEN IF } D \neq d_1 \\
\text{THEN } A \\
\text{ELSE } B \\
\text{ELSE } B
\end{align*}
\]

Notice that the former, (a), has the effect of minimizing the amount of code that is added but it also complicates the condition. In the latter case, (b), the same effect is obtained by introducing a new structure. Here, the individual conditions are simpler, but the code is longer and another level of embedding is required. Over time, new conditions and structures introduced during maintenance make the code increasingly complicated. More important, they increasingly tend to interact with other parts of the code. These interactions get increasingly difficult to manage and, sooner or later, the code becomes so 'fragile' that it essentially becomes impractical to make further changes.

Even worse, perhaps, most existing code contains more than just a modicum of unstructured 'goto' statements. The use of unstructured code can dramatically increase the rate of software obsolescence. This is particularly true with older languages like COBOL and FORTRAN which were developed before the importance of structured programming became well recognized.

INTRODUCTION OF CASE

If used judiciously, traditional Computer Assisted Systems Engineering (CASE) technologies can be of help in designing new systems which may be easier to maintain. Like
documentation, however, even the best-laid system designs get increasingly out of date over time.

Given the current environments in most large MIS departments, it is not surprising that most re-engineering tools deal with COBOL and run on mainframe computers. What these tools do essentially is take unstructured COBOL and replace the 'gotos' with structured constructs.

The input is poorly structured code, which is familiar in varying degrees to the programmers responsible for maintaining it. The output, on the other hand, is well structured code that looks like an entirely new program. During the automatic restructuring process what happens is that new and unfamiliar variables (with arbitrary names) are introduced to allow structured constructs to replace the ubiquitous 'gotos'. This is exactly what Sullivan (Government Computer News, April, 1989) had in mind in his article entitled 'Re-engineering separates program from its owner'. In effect, rather than making maintenance easier, automatic re-engineering in this sense can dramatically increase difficulties. With non-trivial programs it can take a long, long time before one reaches 'break even'.

COGNITIVE ASPECTS OF MAINTENANCE

Accordingly, many of the difficulties in maintaining old code have a cognitive basis. 'Gotos' inherently encourage sloppy thinking. 'Patches' introduced to fix problems or to add new functionality all-too-frequently distort original (cognitive) intent. Automatic re-engineering adds structure but introduces the cognitive overhead of arbitrary, non-meaningful semantics.

In introducing new conditions, for example, it is a cognitive issue as to whether to supplement conditions ((a) above), to increase the level of embedding ((b) above), or to reorganize the code. It is not a syntactic issue. Maintainability depends on the clarity, or meaningfulness, from a cognitive standpoint of the conditions (and operations) involved. The former approach (a), for example, would not be bad where the conditions entering into the complex condition have a clear conceptual meaning. For example, this might be the case with a complex condition like 'smart or funny'. On the other hand, it might be better to replace a complex condition like 'lumber and (three or four)' with the intended and more precise meaning 'stool'. One can envision similar choices between reorganization and adding additional levels of embedding.

Introducing new functionality while maintaining maintainability, however, all too often calls for reorganization. Such reorganization presumes an understanding of the overall structure of a system, and not just the code. In effect, it is essential that one be able to step back and take a broader, more abstract view. Anyone who has programmed, or who has worked closely with programmers, knows how infrequently this is done in practice. By its nature, programming requires careful attention to details. In this context, it is all too easy to lose sight of the 'big picture'. In an analogous situation, good proofreaders actually train themselves to ignore meaning when proofreading text. (Thankfully, proofreading is a disappearing art owing to the introduction of word processors.)

THE NEED FOR ABSTRACTION

What one needs in this context, then, is a system which allows users, system designers and programmers alike to view and modify a system at whatever level of detail, or conversely at whatever level of abstraction, is desired.
It is not sufficient to look at the system as a whole from one perspective (e.g., 'bubble charts'), and the bottom level source code from another. A more uniform perspective allowing one to look at the system at any level of abstraction would be more desirable from the standpoint of reducing cognitive load. (Among other things, uniformity makes it easier to 'chunk' information, something which is well known to reduce memory load (e.g., Miller, 1956; Voorhies and Scandura, 1977).) Moreover, one must be able to view data in this way as well as process. This implies a hierarchical representation where higher levels in the hierarchy represent higher levels of abstraction. Figure 1 shows examples adapted from Scandura (1987).

In light of the above, a cognitive perspective requires a hierarchical view of re-engineering. Specifically, re-engineering existing code from a cognitive perspective involves three major activities:

1. reverse-engineering code into a hierarchical, and preferably visual, environment where the entire system, as well as the code itself, can more easily be analysed and understood;

2. reorganizing system design, along with the code, to better reflect the current hierarchical functionality and semantics of the system; and

3. annotating the hierarchical environment so as to describe system components at various levels of abstraction.

In addition to making a system easier to maintain, re-engineering is often the first step in translation from an older programming language (e.g., COBOL, FORTRAN) into a more modern one (e.g., Pascal, C or Ada). Such conversion is desirable especially where one wants to move to new hardware. Use of the relatively hard to read C language, for example, is often justified because it runs under the widely used Unix operating system and is easily portable to computers ranging from micros to mainframes.

However, then, is the above to be accomplished. Clearly, simply restructuring unstructured code will not suffice. All restructuring does is modify the bottom (code) level of representation.

Those familiar with contemporary CASE tools might think that reverse engineering into 'bubble charts' might serve the purpose. In this context, 'bubble chart' is being used to encompass traditional data flow diagrams, with and without 'real time' extensions, entity-relationship diagrams, and so on. Perhaps we could analyse existing code and extract hierarchical information from it, thereby making it possible to reconstruct 'bubble

---

Figure 1. (a) Hierarchical representation of procedure. (b) Hierarchical representation of data
charts’ from the code. Unfortunately, no one has succeeded in doing this in a general way. About the best one can do is parse the code, extracting useful information and importing that information into a neutral database. From such a database one might, in principle, generate visual bubble chart representations (by introducing positional information).

LIMITATIONS OF ‘BUBBLE CHARTS’

The limitations of ‘bubble chart’ representations in the proposed context are the subject of another paper and will not be developed here. For present purposes it is sufficient to observe the following:

(a) ‘Bubble charts’ look entirely different from corresponding detailed designs and source code. The dichotomy between them is inherent; a schism necessarily exists at some level of analysis.

(b) ‘Bubble charts’ are best used for simple high level overviews of abstract data and processes on that data. ‘Real time’ extensions can accommodate procedural sequence and control issues, but they add a level of complexity at odds with the original goals of visual clarity.

(c) Front end CASE tools display successive refinements of ‘bubble charts’ in different rectangular windows. This not only limits the number of levels which may be seen at one time (inherent in the size of the display screen) but it becomes increasingly difficult to figure out which refinement window goes with which higher level element. Even experienced CASE users find it difficult to find their way around in more than three or four levels of refinement at a time.

The diagrams of Figure 2 correspond to the three basic types of structured refinement (sequence, selection and iteration) and are included for comparative purposes.

OTHER ALTERNATIVES

If we want to represent all levels of abstraction of a system in a uniform way, then we are forced to look elsewhere. Action diagrams and Warnier–Orr diagrams (e.g. Martin and McClure, 1988) are perhaps more viable in this context. They effectively use embedded brackets in one form or another to mark off various levels of abstraction. In principle, one could map successive brackets into a tree-like data structure which could effectively contain higher as well as lower level information.

![Figure 2. (a) Sequence. (b) Selection. (c) Iteration](image-url)
The main limitation of these representations, however, is that they say nothing visually about sequence or control. They are essentially just structure outliners. Information pertaining to sequence or control must be added to the verbal descriptions—thereby confounding abstraction with sequence and control issues.

Nassi–Shneiderman diagrams are better in this respect because sequence and control are dealt with visually. As the examples of Figure 3 illustrate, Nassi–Shneiderman diagrams are easily understood by any analyst, system designer or programmer. Notice that sequential control is handled more naturally than in data flow diagrams. Indeed, recent research by Professor Scanlon (1987) at the California State University shows that 75–80% of students learning how to design systems strongly prefer Nassi–Shneiderman diagrams over English-like pseudo-code.

Since Nassi–Shneiderman charts have been around as long as 'bubble charts', why are they not more widely used? Perhaps the major consideration in this regard is that they are hard to draw and even harder to change. It is much easier to connect bubbles (with lines) in different ways than to make corresponding changes in a Nassi–Shneiderman chart. Today, with the advent of CASE this should no longer be a problem. In principle, Nassi–Shneiderman diagrams might be redrawn automatically just as well as 'bubble charts'.

Another important factor that has worked against Nassi–Shneiderman diagrams is that selections are handled by splitting rectangles vertically (see Figure). This limits severely the number of levels of refinement that can be displayed without having to open a new window. A third major limitation is that Nassi–Shneiderman charts do not deal with data.

FLOWFORM REPRESENTATIONS

As shown in Figure 4 (also see Scandura, 1987), FLOWforms combine the advantages of bubble charts and Nassi–Shneiderman diagrams with none of their disadvantages. FLOWforms have the further advantage of making it possible to see higher- and lower-level abstractions at the same time and in the same context.

Experience with one CASE tool (see below) demonstrates that FLOWforms can be created and changed more quickly than bubble charts. Moreover, informal experiments suggest they are easier to understand than either bracketing systems or bubble charts. Moreover, modifications can be made at any level with direct visual traceability to all effected levels of abstraction. That is, as shown in Figure 4, one can describe systems in English and in context at whatever level of abstraction is desired.

FLOWforms constitute the basic building blocks of systems that can be of arbitrary size. Higher level FLOWform modules contain links to the other FLOWform modules they use, much as higher level routines are related to the subroutines they call. In this regard, it is important to note that changes in one FLOWform do not propagate to other FLOWforms, unless one explicitly changes links in the 'calling' FLOWform. This has important benefits for such things as configuration management and version control which are addressed elsewhere.

---

2 A formal study is currently under way at the University of Pennsylvania.
**RE-ENGINEERING FROM A COGNITIVE PERSPECTIVE**

With this background, let us return to re-engineering. Rather than forcing a choice between redevelopment and salvaging old code, the CogApp to systems engineering and re-engineering puts new development and maintenance on the same plane. In this view, the choice is not either-or but how much new development and how much revision. More precisely, CogApp involves extracting the functional 'essence' of the current (or planned) system. Capturing functionality in this sense involves knowledge engineering carried out at a relatively high level of abstraction, independently of how the code is actually structured.

Based as it is on a particular form of knowledge engineering, called structural analysis (e.g. see Scandura, 1982, 1984, 1987; Scandura et al., 1974), CogApp requires one to look at both data and process from the top down. From a cognitive perspective, it makes no sense to analyse data before considering process or process before data (although, of course, some kinds of systems involve relatively more data, and others more procedure). It is important at each stage of analysis to have a complete picture, or cognitive map, of the system at some level of abstraction. Ideally, one would want to test high level system designs independently of the existing or planned code.

Conversely, structural analysis assumes availability of certain basic capabilities that have meaning in the context of the desired system. If the system involves maintaining a business system, for example, one meaningful component might be making out an invoice. If the system were to control an airliner, then reasonable software components might detect air pressure or control various functions. In dealing with an existing system, this phase involves analysing the code to determine whatever meaningful components might exist therein.
The CogApp involves three basic steps:

(1) Create a high level or logical design which captures the functionality of the current or planned system. This may be done initially at a *very* high level of abstraction as in developing any new system. Where the planned functionality of the system is not fundamentally different from current functionality, one can typically move rather quickly to a high level design. To make this approach work it is highly desirable that the design be tested empirically to see if expectations as to flow of control (and/or displays) function as desired. (The basic approach is of the same genre as those described by Boehm (1988) and Yeh (1990, Ch. 1) but with different emphasis.)

(2) Reverse engineer existing code to extract the basic system design so it can be more easily analysed, evaluated, documented, restructured and/or otherwise modified. The essential aspect is identifying meaningful components that might reasonably be 'cleaned up' and reused. Particular attention is paid to aspects of the code that correspond to components of the high level design.

(3) Map appropriately documented and modified low level components into the high level design. Various adjustments such as renaming parameters or variables should be made in the process but always with the goal of preserving the desired high level design.

This approach has two important advantages:

(a) Functionality of the revised system is reflected directly in the new higher level code. This makes it relatively easy for both new and old programmers to modify and otherwise maintain the code.

(b) As much of the existing code as possible is retained (without distorting the desired system) thereby reducing redevelopment costs—sometimes dramatically.

In order to efficiently implement CogApp, we need a system development and maintenance system with a variety of features. First of all, we need a way to develop and execute high level designs independently of the underlying code. As I have pointed out in many public presentations, testing designs at a high level of abstraction can dramatically reduce the number of necessary test items. When all testing is done after a system has been implemented, the number of to-be-tested paths increases exponentially with system complexity—the well known combinational explosion problem. By way of contrast, when testing is done from the highest levels of abstraction, the number of required tests increases only additively with system complexity. The order of magnitude of improvement in the latter case is dramatic, turning an impossible task into something which is quite feasible.

We also need a way to take our existing source code and reverse-engineer it into hierarchical FLOWforms. It would be nice, in addition, if we could eliminate in the process such messy notation as ‘BEGIN’, ‘END’, ‘,’ ‘‘;’, ‘;’, ‘;’ and so on—leaving only the basic pseudocode that tells humans what the systems actually do. Naturally, in doing so we would not want to give up any of the flexibility or power of the original language (see Figure 5).
It would be even nicer if our ideal system also could automatically construct English descriptions at an appropriate level of abstraction by simply analyzing the code. Given the current state of research, we are far from this ideal. For the foreseeable future, we will have to settle for making use of existing program documentation and inserting it at potentially relevant places in the abstraction hierarchy. We would, of course, also need to be able to change the ‘higher level annotation’ at any time and in any way we want—and to be able to do this directly in the visual FLOWform environment.

We have already suggested another key requirement. Once the source code had been reverse-engineered into visual FLOWforms, we would want the ability to modify the FLOWforms as we wish at any and all levels of abstraction, easily and with complete flexibility. Since we are working with pseudocode in a hierarchical FLOWform, of course, we also would need the ability to regenerate source code for the systems we modify.

Finally, if we wish to port to a new environment, we might also welcome the ability to convert from one pseudocode language to another (e.g., from COBOL to C)—all the while retaining the original high level system specifications.

**PRODOC re/NuSys Workbench™**

The PRODOC re/NuSys Workbench™ is a full life cycle design, development and maintenance system well suited for renovating existing systems. PRODOC™ uses FLOWforms to represent systems at arbitrary levels of abstraction in a highly interactive visual environment. Among other things, the use of FLOWforms helps eliminate representational inconsistencies and awkward transitions between analysis and design.

System specifications and high level designs can be tested at any level of abstraction. For example, English specifications (including sample screens) can be executed to check the flow of control; high level designs, to test underlying logic. High level designs, in

---

**Figure 5**

---
Figure 5 continued

return, may be refined systematically into directly interpretable prototypes using PRO-DOC™'s high level prototyping language (which allows both procedural and data (object-oriented) execution). In this case, source code can be generated automatically from working prototypes ready for the compiler. When linked with a run-time library, prototypes can be run as separate applications.
Alternatively, high level designs can be translated automatically into any of the languages supported by PRODOC™: Pascal, C, Ada, COBOL or Fortran. In this case, the user must create and/or have available for reuse functional routines corresponding to the components of the high level design. Maximum flexibility can be obtained by implementing low level routines in FLOWforms containing language-specific detailed designs and Pascal, C, Ada, COBOL or Fortran pseudocode (or full source code in any language such as assembler). Pseudocode support includes syntax, consistency and redundancy checking along with automatic declarations and full source code generation. Existing source code (and export files from other tools) can be reverse-engineered (i.e., automatically imported into properly structured FLOWforms) at roughly the speed of a compiler. Another feature, useful in converting from old to new environments, is PRODOC™’s ability to automatically translate between pseudocode languages.

Front-end and back-end databases in most CASE systems are large and distinct. PRODOC™, on the other hand, uses a distributed database consisting of modular FLOWforms with links between them. Individual FLOWforms contain all abstraction levels (within a system) relevant to that module. Modifying one FLOWform leaves everything else as before, thereby facilitating configuration management. Automatically traceable, immediately visible documentation is a natural byproduct of the hierarchical FLOWform specification and design.³

A SHORT CASE HISTORY

Motivated by the results of 20 years of basic research in the cognitive and computer sciences, PRODOC™ development was begun in 1983. By early 1984, we gradually began to use PRODOC™ in its further development. Since 1985, all further development and enhancement has been done exclusively within PRODOC™ itself.

By mid-1986, even the early ‘bootstrapping’ routines had been reverse-engineered into PRODOC™. Therein lies the genesis of CogApp re-engineering.

We had almost 100,000 lines of bootstrapping code, about half of which related to PRODOC™’s dynamic simulation and prototyping facilities. This code was functioning correctly and it was doing essentially what we had wanted at the time. Like most production code, however, the code had gone through more than its share of revision. Patch after patch had left that part of the system extremely fragile. It was a major chore to make even the most trivial changes for fear of introducing unanticipated interactions. Introducing major enhancements on our ‘wish list’ was unthinkable.

Even though the simulation and reverse-engineering facilities were not nearly as advanced as they are today, they were sufficient to support the basic CogApp to re-engineering. What we did first was to reverse-engineer the original code. The results were dramatic once pseudocode was extracted and ‘uploaded’ into FLOWforms. The structure of the pseudocode still mirrored the original design. Still, the visual representation of that design made the overall organization clearer and what had been unthinkable became at least possible.

As is typically the case, the lower level routines were not in bad shape. In fact, many were quite good. Adding higher level annotation in those cases improved the situation

³ PRODOC™ currently runs on under MS-DOS. A Unix version is available for Sun Workstations with other Unix ports expected by the mid-1990.
even more. With today’s technology, it takes PRODOC™ about two days to reverse-engineer 50,000 lines of code. With the rudimentary version available in 1986, it took about three weeks.

As in similar situations, however, the current functioning of the system and the current organization of the code were largely (if not totally) out of sync. To better understand the current system, we used PRODOC™ FLOWforms to model the then current functionality of the system (with an eye to planned extensions). The design was tested dynamically for verification purposes.

It was at this point that we decided to gamble. Rather than try to salvage the original code, we took the new design as the starting point and mapped the reusable lower level routines into it. Aside from renaming variables and the like, we were surprised (actually amazed) that the entire ‘mapping’ process took only two days plus another half day of debugging. It is important to emphasize, however, that we had been careful in planning and testing the new design, and in fully understanding the reused lower level modules.

The PRODC reNuSys Workbench was only released commercially (after beta testing) in the autumn of 1989. Hence, it is too soon to report on extended results by other users. We have used the CogApp, however, on various development and re-engineering projects—ranging from the development of PRODOC™ itself to the development of intelligent tutoring systems. PRODOC currently is being used in a number of salvage operations, with experience to date very much in line with the case history reported above.

References


