A Cognitive Approach to Reengineering

Joseph M. Scandura
IMS-Scandura

This article proposes a cognitive approach to system maintenance that can dramatically improve systems and minimize renewal costs. This approach lets engineers model and test the structural and functional essence of a system at a high level of abstraction, increasing specificity until contact is made with available data and computational resources.

Saddled with large quantities of essential programs written in obsolete languages such as COBOL or FORTRAN, Department of Defense (DoD) contractors and agencies are faced with a series of unpalatable choices. One option is to merely continue with the same old software, patching it where possible to meet the most pressing needs. However, reengineering can add a measure of efficiency previously unattainable when properly implemented. Although renewal costs can often be recovered over time, initial costs are often prohibitive.

Because of this dilemma, reengineering, reuse, and maintenance are hot topics in software engineering. However, there is still much confusion about what reengineering involves and even more concerning the benefits of current reengineering tools.

Approaches to Software Reengineering

Several classes of reengineering tools have evolved over the past few years, offering software engineers an array of choices. Each of these tools has strengths and weaknesses, and their effectiveness is largely affected by how they visually represent the physical aspects of the system, particularly when converting to higher order languages.

Code Analysis Tools

Code analysis tools help gather useful information about existing systems, such as complexity measures, calling hierarchies, cross-reference lists, and information about the organization of code (or the lack thereof). Although they provide insightful information about a system, their major limitation is that they do not provide programmers with the information they need to find the source of problems or to properly change the code.

Restructuring Tools

Certain kinds of code modifications can be automated. Some commercial tools can automatically restructure FORTRAN and COBOL source code, for example. With FORTRAN, restructuring is almost always desirable. Eliminating GoTos significantly facilitates code understanding from a cognitive point of view. Instead of having to scan and integrate scattered code fragments, engineers have immediately relevant processes available in one context.

COBOL restructuring poses a unique set of problems because GoTos cross module (paragraph) boundaries. Eliminating GoTos and fall-throughs generally results in what amounts to entirely new COBOL programs [1]. Nonetheless, if the goal is long-term maintainability, COBOL restructuring is an essential first step.

Design Recapture Tools

A second, somewhat newer class of tools is concerned with design recapture—analyzing source code to determine and visually represent relationships between source code modules. Typically, the information obtained is represented in some type of structure chart or module hierarchy.

The basic technology generally involves simple parsing techniques in which modules are identified, and attendant relationships are captured for later visual representation. The process is not unlike extracting a table of contents from a book. It is clearly worthwhile to extract overall relationships within a system and represent them in a visual environment where they can more easily be modified. However, programmers cannot rely on overall structure for making changes to code. As any programmer knows, the devil hides in the details.

Most reengineering tools provide limited access and editing tools to access and modify module code. However, such tools do not solve the biggest problem: understanding details in order to know how to modify the actual code.

Cognitive Approach to System Renewal

First used to construct representations of human knowledge, structural analysis has equal applicability in software engineering, offering an integrated cognitive approach to system development, reengineering, and conversion. Structural analysis shares certain attributes with B. W. Boehm's spiral model [2] and R. T. Yeh's programming by design [3], and involves modeling and testing a system's structural and functional essence at a high level of abstraction, increasing the specificity until contact is made with available data and computational resources.

An essential characteristic of this method is that both data and process must be represented at the same level of abstraction, analogous to the representation of human knowledge [4,5]. The more knowledgeable the human population being modeled, the more abstractly the knowledge can be represented. Put differently, the larger the chunks or atomic rules, the more easily any system (or body of knowledge) can be modeled [6, 7]. Testing and diagnosis of individual knowledge at higher levels of abstraction is more efficient because fewer paths are involved [4].

Testing and debugging software designs makes sense only when data
and process are represented at the same level of abstraction. Structured analysis, with its emphasis on process, is inadequate because it neglects the importance of data. Conversely, to defer the representation of processes in information engineering would leave no actions to test. Object orientation has limitations in regards to testing behavior and operation abstraction. My research has shown that operations within objects are not the same as operations on objects. The former corresponds to automatic human perception, the latter to conscious cognition [8].

The above ideas are implicit in the method of structural analysis [5,9]. Although initially use to analyze cognitive processes, structural analysis has been used successfully in software engineering. The process is essentially the same whether structural analysis, that is, the cognitive technology is used to design and develop new systems or to reengineer old ones. In the former case, the to-be-developed system exists only in the mind of the analyst, designer, or end user. In the latter case, we begin with a fully functioning system. Both cases make heavy use of reusable routines (with

systems) or of code salvaged as a result of reengineering. This is true whenever code is salvaged from a legacy system, whether as is before or after conversion to Ada or C++.

System renewal involves modeling the behavior of a designed system from the highest levels of abstraction. In parallel, existing code is reverse engineered into a modular repository compatible with that model. The language-independent model and the reverse-engineered code may optionally be converted into the same target language, for example, C, C++, or Ada. Finally, the debugged model is linked to reusable modules in the legacy code. The reused code is supplemented as necessary with other libraries or new modules [10,11,12].

### Module Visualization

Source code is not easy for the neophyte to understand. Cognitive studies show that even skilled programmers spend only 10 percent of their time making changes to source code—the other 90 percent is spent trying to understand the code. Consequently, anything that can facilitate understanding will pay handsome dividends.

"Pretty printing," though a step in this direction, is not sufficient. Action diagrams [13] organize code structure and make it easier to perceive structure groupings by bracketing code. However, it remains for the programmer to distinguish different types of structures and to separate relevant code from irrelevant detail. Action diagrams still contain extraneous syntax and bracketing information (such as,, ], }, ;, Begin, End); studies show that irrelevant information, even when highly familiar, increases cognitive strain, which lowers the capacity for productive thought. The visual bracketing that is provided for different structures in Flexforms can improve comprehension. Flexforms are based on a unique contextual display process (U.S. Patent 5,262,761) in which each type of procedural refinement, e.g., sequence, selection, loop, has a perceptually distinct form, helping engineers better understand the structure and content of a legacy program. Flexforms are highly dynamic and flexible in appearance, making it possible to represent procedural logic at any or all levels of abstraction, from highest to lowest. Programs based on module visualization principles aid human comprehension by representing structure visually and eliminating irrelevant detail.

Although most experts recommend limiting the cyclomatic complexity of individual modules to no greater than 10, C modules with a cyclomatic complexity as high as 297 are being successfully maintained in Flexforms. It is unlikely that these modules could be understood by looking at the corresponding source code in ordinary text files, let alone maintained as such. Complexity this high is not recommended, of course, but because such modules can be maintained at all illustrates the value of Flexform representation (Figure 1). Compared to ordinary source files, estimated savings in maintainability range between 25 percent and 50 percent.

Although better visual representation helps, this alone is not sufficient. To improve on code analysis tools, it must be possible to automatically construct such visualizations from code and to modify the code directly in the visual environment. In this context, an integrated reengineering workbench should automatically reverse engineer existing code into pseudocode Flexforms. The tool should also provide an interactive environment in which Flexforms can be edited, documented, restructured, and customized to support multiple environments and should regenerate full source code as desired.

### Contextual vs. Separate Windowing

Two kinds of representation are implicitly described above: representation of relationships, e.g., structure charts, and representation of modules. Rectangles and circles connected by lines (bubble charts) are commonly used for these purposes. However, they are inadequate for other purposes. Several views, e.g., dataflow and control flow, are needed to capture the essence of many complex systems, especially real-time systems. Because each view represents only part of the information contained in the code, it is often difficult...
to modify many aspects of the code. However, Flexforms can represent dataflow diagrams, structure charts, control-flow, and entity-relationship diagrams and context diagrams [10].

It is important to distinguish the different kinds of information represented, because each deals with different system aspects, such as modules, module relationships, and file/unit relationships. Certain kinds of representation are best displayed and edited in separate windows. For example, structure charts convey information about relationships between modules, which is intrinsically different from the modules. The same difference exists between modules and the relationships between compilation units or files.

However, separate windows are not desirable when dealing with different levels of abstraction within the same module. To display this information in different windows places severe restrictions on the number of levels that can be displayed on a screen thus affecting ease of comprehension. Expansion in different windows, for example, makes it difficult to remember which windows (expansions) go with which elements in other windows. It is well known in cognitive psychology that the number of different chunks of information that a human can deal with simultaneously approximates the magic number 7 plus or minus 2 [6,7]. However, most experts agree that it is difficult to understand more than three or four levels of a data-flow diagram at one time.

One solution is to use some form of contextual windowing. Flexforms accomplish this by allowing explosion directly in context. Lower level detail is automatically displayed within the element that contains it. Thus, Flexform rectangles can be expanded without affecting the context above or below. This lets programmers see more detail without losing the general picture.

Graphic elements, such as boxes or circles connected by lines, lack this feature. When you expand an element in a bubble chart it merely changes the overall scale. In most tools, the original context disappears off the monitor [10]. In contrast, Flexforms let the user view within module relationships at any desired level of abstraction (Figure 2) as well as relationships between modules, units, and even programs. Similar attempts to solve this problem include displaying connections between original windows and "fish eye" type expansions. These retain relationships to some extent but require more screen space and can hide essential information.

Figure 2 shows several levels of a Flexform hierarchy with cursor positions corresponding to different levels of the defined tree-like structure. The number 1 corresponds to the top-level structure, 2 and 3 correspond to 1's child structures, 4 to the loop body (child) of 3 and 5, and 6 to children of 4. All except 6 show terminal elements only. Comment within 6 refers to the tree element immediately above the terminal elements of 6. Higher-level elements are displayed inside a distinguishable border (a dotted frame). Fanning in and fanning out in tree-like structures is accomplished via visually distinguished clones, but none is shown in the diagram.

Of course, different kinds of representations, for example, modules or call hierarchies, are represented in different windows. Flexforms differ from other views in that the same intuitive representation is used throughout. Moreover, the user never has to look at code per se, even at the lowest levels of module detail. All desired modifications can be made directly in Flexforms without restriction—any program that can be written in a higher-order language can be written directly in that language's Flexform.

Conversion to New Environments

Because of continuing improvements in hardware and operating systems, engineers are often faced with converting old software to new environments. To make matters worse, engineers frequently must simultaneously maintain two or more versions. Normally, this is accomplished by separate teams of programmers maintaining two or more sets of files. Obviously, it would be better if the same engineers could maintain multiple versions in one set of files.

One solution is to use conditional compilation metacommands supported by most compilers. This approach, however, clutters the code and makes it increasingly difficult to read and understand program logic. Meta-commands also provide extra noise, which lowers comprehension.

However, Flexforms minimize the amount of irrelevant information by labeling structures that are unique to a given platform or operating system. These labels are used during code or report generation to automatically produce multiple versions for different environments on demand. Consequently, only the Flexforms need to be preserved.

Conversion Between Languages

When converting code into a different operating platform, for example, from IBM's MVS to PC-DOS or UNIX, it is often desirable to use this opportunity to convert the code into a more modern programming language such as C++ or Ada. Some argue that the only reasonable way to do this is to rewrite the code. However, this approach is costly at best, and the level of programmer likely to be assigned to this task will probably produce code that is not much better than the original.

A second approach involves source-to-source translators that directly convert source code in one language into another. Many argue that such tools merely produce poor code with few real improvements—"Ada2BOL" or "AdaBOL," for example. A third approach converts the source language into an intermediate
form that features semantic and syntactic characteristics; code for the target language is generated from this intermediate language. This often creates better results, because the method makes it easy to deal with semantic and syntactic issues. However, the overall process tends to be slower and more complex, and high-level reengineering issues tend to go unaddressed.

A fourth approach refines the above technique by more sharply distinguishing syntactic and semantic aspects of a translation. Parsing technologies rapidly complete syntactic aspects of the conversion, usually up to the individual statement level. They also can be adapted to map multiple statements, for example, FORTRAN Format and Print statements, into the target language. Semantic issues are addressed by slower, yet more powerful semantic transformations. Semantic transformations become increasingly necessary as the mappings address deeper, more abstract semantic differences, for example, FORTRAN Commons vs. Ada packages.

What most differentiates this fourth approach is that the conversion works both syntactically from the bottom up and semantically from the top down—a truly cognitive approach. Bottom-up analysis corresponds to the largely automatic processes a skilled programmer uses in line-by-line conversions. Top-down analysis corresponds to the more thoughtful analysis that goes into making high-level design decisions. The results of such automated conversions can approach or even exceed those performed by an average human programmer. There are, of course, practical limits to what any generic translation tool can do, especially considering the numerous variations of languages, language definitions, compilers, operating systems, and libraries. Consequently, a full solution to the conversion dilemma must lend itself to customization.

Automatic conversion can be accomplished in two steps. The first involves reverse engineering the source code, for example, FORTRAN, COBOL, or C, into a modular, object-oriented Flexform repository. (This modularity makes the Flexform repository ideally suited for client-server environments in which information may be scattered.) Once reverse engineered, the semantics, as well as the syntax of the source code, are directly accessible. (Reverse engineering normally is fully automatic, although minor customization may be necessary with nonstandard code.)

The second step involves conversion. Visual Flexforms that contain pseudocode in one language are converted into Flexforms that contain pseudocode in another language. Parsing techniques perform the simpler syntactic conversions. In turn, semantic post-processors can turn the translator output into a “good” Ada or C++. Good in this sense means that it uses Ada constructs, for example (such as packages), that have no direct counterpart in the source language. With appropriate customizations, the results would be indistinguishable from those produced by an Ada expert.

Both the syntactic and the semantic post-processing aspects are customizable. The basic machinery also is extendable to new languages. Currently, C, C++, and Ada conversions from FORTRAN, C, Pascal, and COBOL are supported. From 90 percent to over 99 percent of the code is converted automatically, with higher-level designs preserved in the process.

These levels of automation may be further improved through customization, which is straightforward once engineers decide exactly what must be done and under what circumstances.

To summarize, code translation is usually wise when the software can be better maintained in the new language. The DoD recognizes that Ada programs are easier to maintain than programs written in FORTRAN, COBOL, C, or Jovial. The bottom line is: source-to-source and intermediate-language translators represent a reasonable approach if no further code maintenance is desired. But if this were the case, why translate the code to begin with?

System Redesign

For situations that call for continual code updates and enhancements, there is an implied need for customizable solutions that make explicit provisions for reengineering. But although many situations seem to call for creating entirely new or renewed designs, in most cases it is possible to salvage much reverse engineered or converted code rather than build an entirely new system.

Reusable code can be either highly specific or relatively comprehensive. In most cases, it should be relatively modular. To reuse code from an existing system to build a better system in the same domain has the major advantage that large, high-level modules can often be reused in implementation. Our experience suggests that, at a minimum, 50 percent to 60 percent of existing code—and usually much more, sometimes as much as 99 percent—is reusable in redesigned systems. The key to reusability is not simply the quality of code—its proven stability over time can be even more important. As long as the “black box” works and is not likely to change, there is little need to look inside.

Testing New Designs

Most front-end computer-aided software engineering tools support new design and some also support simulating display and input screens, largely to ensure user satisfaction. Both factors play an important role in system design or redesign, but they are not the only factors. Confidence in a new design comes only from testing and debugging underlying logic.

Testing at the design level requires some form of executable specifications, which unfortunately often require learning an entirely new language—a hindrance that can greatly reduce overall benefits. Formal specification languages are required largely because the commonly used design methods favor either data analysis or process analysis. Lacking a balanced approach to data and process, they do not lend themselves to debugging designs.

A cognitive approach to systems design demands that data and process be considered in parallel. Arbitrarily abstract specifications may be used with respect to both data and process. “Destroy (missile),” for example, is as adequate a specification from a cogni-
Interfacing Renewed Designs
Creating and testing a high-level design is only one part of the problem. The high-level design must be interfaced with reverse engineered or otherwise reusable code. Industry's current solution is to convert high-level designs to the target language and to create an interface between converted designs and the reusable code. Checking processes in turn provide an interactive, semiautomatic way to create links between converted designs and the data and process resources referenced in those designs. In contrast, source code generated from pseudocode Flexforms and only a handful of other tools can be compiled and linked directly to the reusable code.

Automating System Redesign
Another way to improve a software system is to manipulate system semantics under program control. Flexforms facilitate this process because they directly expose program semantics, thereby facilitating such manipulations. For example, in any given application, the database is typically accessed in a relatively small number of different ways. Each access method corresponds to a semantics-based pattern that, e.g., with an appropriate set of semantic-based routines, can easily be detected and automatically modified, e.g., to support a new database.

An Ada conversion process is illustrated in Figure 3. Figure 3a shows sample FORTRAN code, and Figure 3b shows the FORTRAN after restructuring and reverse engineering into a Flexform. The structure of the procedure, including control flow, is immediately apparent without special training. As noted earlier, further benefits derive from the ability to collapse or expand Flexform structures directly in context.

An example of parsing the FORTRAN Flexform is shown in Figure 3c. In this case, the FORTRAN code is so simple that the translator does all of the work directly. The translator also handles more complex procedural constructions. FORTRAN Print statements, for example, are key to associated but possibly distant Format statements. Both are needed to determine the corresponding higher-order language output statements.

Semantic post processing is usually a necessity. FORTRAN arrays, for example, are initialized in Data statements, which are separate from variable and type declarations. These are defined in Ada as arrays, but with initializations in procedural sections.

Most FORTRAN-to-Ada complications, for example, reside in the data, and these problems are usually handled during semantic post processing. The semantic post processor also deals with program- and system-level issues. For example, common variables in FORTRAN are global. In this case, the FORTRAN-Ada Semantic Post processor puts these in an Ada Package Flexform, with appropriate references in modules using these resources. Similarly, FORTRAN Procedure Flexforms convert into top-level Ada Procedure Flexforms. The translated subroutines called therefrom are automatically inserted into Package Flexforms and corresponding Package Body Flexforms.

A wide range of reengineering tasks can be accomplished automatically by building custom tools with Flexsys's meta-facilities. Higher-order language source code is generated automatically by merely selecting the proper option.

Once a new design is acceptable, it can automatically be converted into a higher-order language. Procedural code in high-level design Flexforms is automatically converted into Ada Flexforms. The final step involves linking the Ada Design Flexforms to reusable modules. The user must classify each identifier introduced in the original design, e.g., as either a variable or a function. In the case of variables, the user must also specify associated types in the reusable code.

The process is essentially the same with all legacy code. Code is reverse engineered, translated, and semantically post processed. Optional redesign or customization or both comes next. Finally, the full Ada code is generated. There are, of course, special considerations. Unlike C, Ada does not support pointers to functions, so a custom strategy must be devised in C to Ada conversions. Similarly, structured COBOL typically requires explicit Exits (to avoid fall-throughs). These Exits are
generally converted into Ada exceptions. Other languages require customization of both the syntax-oriented translator machinery and the semantic post-processor. ♦

About the Author

Joseph M. Scandura, Chairman of IMS-Scandura, is the author of over 175 scientific publications including 8 books. In addition to software engineering, he has published articles in psychological, artificial intelligence, mathematics and instructional research journals. He has presented papers and appeared on panels at numerous national and international conferences. His focus is on applying cognitive principles to software engineering and reengineering.


Scandura has a bachelor's degree and a master's degree, both from the University of Michigan. He also has an honorary master's degree from the University of Pennsylvania and an honorary doctorate degree from Syracuse University. He was a professor at the University of Pennsylvania for 25 years, is a fellow of the American Psychological Association and appears in Who's Who in America.

IMS-Scandura
1249 Greentree, Suite 100
Narberth, PA 19072
Voice: 610-664-1207
Fax: 610-664-7276
E-mail: jms@pobox.upenn.edu

References