New Directions in Software Engineering

Joseph M. Scandura, Ph.D.
University of Pennsylvania
and
Scandura Intelligent Systems

Saddled with obsolete code, software managers are faced with a series of unpalatable choices. Managers all too often take the path of least resistance. Whenever possible, existing software is simply patched to meet the most pressing needs. Alternatively, the old software is trashed and replaced with entirely new systems.

Neither approach is ideal. The costs of new development are frequently prohibitive and continuing to patch old code rarely yields satisfactory results. Problems invariably get progressively worse over time.

Several classes of re-engineering tools have evolved over the past few years which appear to offer better choices. Indeed, re-engineering today is one of the "hottest" topics in software engineering. There is a good deal of confusion, however, as to just what re-engineering involves, and even more so as to benefits of the kinds of re-engineering tools that are currently available. In this paper, I shall review analysis, design recapture and system redesign tools, along with the major strengths and limitations of each type. Several tools are mentioned where appropriate, with special reference to the re/NuSys Workbench™.¹

Code Analysis.-- One class of tools involves the analysis of code. Analysis tools are used to determine the complexity of existing systems, and to provide calling hierarchies, cross reference lists and other information concerning the organization of code (or the lack thereof). Restructuring tools and pretty-printers fall in the same general category. They either gather information about existing systems and/or simply

¹ The re/NuSys Workbench is a full life cycle software design, development and maintenance system from Scandura Intelligent Systems, 1249 Greentree, Narberth, PA 19072.
represent that information in better form. In each case, the results are referenced by programmers in modifying code.

Clearly, analysis yields positive benefits. Tools in this category, however, have a major limitation. Information about a system and the corresponding code are essentially separate in analysis tools. One can gain insights or information from the analyses, but programmers still have to find the source of those problems and ways to fix them. Other tools, (e.g., editors) are needed for this purpose.

*Design Recapture.* -- 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 calling hierarchy. The basic technology generally involves simple parsing techniques in which modules are identified and attendant relationships captured for later visual representation. The process is not unlike extracting a table of contents from a book. That is, one looks for headings and similar kinds of information and extracts that information from the body. It is widely recognized that most MIS/DP expense goes into maintaining existing software. Consequently, an increasing number of vendors have begun to introduce tools designed to recapture calling hierarchies -- typically from old COBOL.

Extracting overall relationships within a system and representing them in a visual environment (where they can more easily be modified) is clearly worth doing. Unfortunately, one cannot rely on overall structure in making changes to code, especially where the existing code is poorly designed. As any programmer knows, the "devil hides in the details." Consequently, some re-engineering tools provide limited access to module code. They may eventually also provide direct access to editing tools with which one can modify such code. This approach, if and when it is actually realized, will still leave the biggest problem -- understanding details in order to know how to modify actual code. The kinds of representations (e.g., "bubble charts") used to represent high level designs do not lend themselves well to this task.
Module Visualization.-- Solving this problem requires an interactive, visual environment for representing not only overall relationships, such as structured charts, but the structure of individual code modules themselves. This, in turn, requires better ways to visualize code. The purpose of such visualization is to aid human comprehension both by representing structure visually and by eliminating irrelevant detail. Action diagrams (e.g., Martin 1988; Scandura 1990) help to organize code structure. They only bracket code, however, leaving the human to distinguish different types of structures and to separate relevant from irrelevant detail. Moreover, they are applicable only to low level code. FLOWforms (Scandura 1987, 1990), on the other hand, provide visualization at both levels -- overall relationships and individual modules. In Scandura Intelligent Systems’ re/NuSys Workbench™, existing code is reverse engineered automatically into pseudocode FLOWforms, where the pseudocode can be edited, documented, restructured, customized to support multiple environments and used to regenerate full source code. Analyzing existing source code and representing it visually in this manner requires much more sophisticated parsing techniques than simply recapturing designs (i.e., relationships between modules).

Contextual vs. Separate Windowing.-- The ability to represent system information visually at these two quite different levels, switching quickly between them as desired, makes it possible to maintain complete systems in one uniform visual environment. In this context, FLOWforms have an advantage over other notations. Although switching between different types of representations (e.g., modules and overall relationships) is best done in separate windows, the use of separate windows is not always desirable. When working either at different levels of a calling hierarchy, or when working at different abstraction levels within an individual module, separate windows make it difficult to remember which windows (i.e., expansions) go with which elements in other windows. FLOWforms avoid this problem by allowing "explosion" directly in context. Lower level detail is automatically displayed within the element which contains it. Thus, FLOWform rectangles may be expanded without in any way affecting the context above or below that in which they exist. This makes it possible to
see more detail without losing the general picture. This is not possible using graphic elements, such as boxes or circles, connected by lines. Attempting to open a visual element in this type of representation would simply change the overall scale. Consequently, the original context would quickly extend beyond the bounds of the monitor screen.

**Customization.** -- The ability to switch between overall relationships and module detail, and to make modifications at all levels of abstraction, has obvious advantages as regards maintainability. Nevertheless, no one type of relationship will be sufficient in all cases. Calling hierarchies, for example, usually include references to corresponding parameters. But what about global variables? Or, routines exported from one file or compilation unit to another?

What is needed in this case is a way to determine the kinds of relationships to be captured. It would be desirable if the user could cost-effectively create customized representations of his own. The re/NuSys Workbench™’s checking, simulation and high level design generation capabilities provide one such solution. Reverse engineered modules can be checked interactively to more precisely categorize identifiers and their definitions and/or declarations. Then, higher order routines can be constructed that operate on lower order FLOWform routines and build new FLOWforms more precisely representing the desired relationships.

**Multiple Environments.** -- Another type of customization involves the ability to support multiple environments in one set of files. Tools from NETRONCAP and Scandura Intelligent Systems support multiple environments in this sense. The NETRONCAP tool is designed for use with COBOL, whereas Scandura’s re/NuSys Workbench™ supports C, Pascal, Ada and Fortran as well as COBOL. In FLOWforms this is accomplished by simply labeling structures which are unique to a given platform or operating system. These labels are referenced during code and/or report generation.

**Conversion Between Languages.** -- In moving to a new environment (e.g., from MVS to DOS or Unix), it is often desirable to convert from an existing
programming language into a more modern one. One approach involves the use of source to source translators. These tools take source code in one language and convert it directly into source code in another. This represents a reasonable approach if no further maintenance on the code is desired. In this context, however, one might reasonably ask: "Why translate the code to begin with?" The purpose of translating from one language to another usually is because the software can be better maintained in the new language than in the old. Consequently, it would be best if the conversion were done in an interactive, visual environment -- for the reasons detailed above. Visual FLOWforms containing pseudocode in one language, for example, might be converted into pseudocode FLOWforms containing another language. In particular, the re/NuSys Workbench™ supports translations from popular older languages to most newer languages: COBOL or Fortran to Pascal, C or Ada; Pascal to C or Ada, or C to Ada. From 90 to 99% of the code is converted automatically. Higher level designs are preserved in the process.

System Redesign.-- While each of the above capabilities contributes to overall maintainability, all are based on a common assumption -- namely, that the design of the original system is worth recapturing. This raises the following questions: Given a poorly designed system: "Why would one want to capture the design?" Or, if the code is bad, "Why would one want to translate it?"

Many situations call for creating an entirely new or renewed design. Rather than having to build an entirely new system, however, it is possible in some cases to salvage code from the original source by reverse engineering. To be a candidate for reuse, the code may be either highly specific or relatively comprehensive. In most cases, however, it should be highly modular. Perhaps the major advantage of reusing code from an existing system to build a better system for the same or a similar purpose is that large high level modules can often be reused in implementing renewed designs. Experience suggests that, at a minimum, 50 to 60% of existing code, and usually much more -- to over 99%, is reusable in redesigned systems.
Most front end CASE tools support new design. Some also support simulating display and input screens, largely to insure user satisfaction. Both of these factors (i.e., design and displaying user screens interactively) play an important role in system design or redesign. However, they are not sufficient. Confidence in a new, high level design comes only from testing (and debugging) the underlying logic. Such testing can be done only where both data and process are represented in the design, at the same level of abstraction. The design methodologies commonly used in traditional CASE tools favor either data analysis (e.g., information engineering) or process analysis (e.g., structured analysis). Lacking a balanced approach to data and process, they do not lend themselves to debugging designs.

The inability to test the logic underlying high level system designs prior to implementation is a major limitation. As Scandura (1990) demonstrated, the number of tests required goes up exponentially with complexity if all testing is done after implementation, whereas the number of tests required only goes up additively if testing is done from the top down. In the example cited, the number of empirical tests required in a rather simple system was on the order of $2^{100}$ if one waited until complete implementation before testing. On the other hand, only about 300 tests were required where testing was done successively from the highest levels of abstraction.

**Interfacing Renewed Designs.**—Creating a high level design and testing it, of course, is only one part of the problem. Another involves the interface between high level designs and reverse engineered, or otherwise reusable code. One solution would be to convert high level designs to the target language and to create an interface between converted designs and the reusable code. To my knowledge, the re/NuSys Workbench™ is the first and, to date, the only CASE and re-engineering solution that explicitly supports the entire process. High level designs are first translated automatically into pseudocode FLOWforms in the target language. (Source code can be generated from such FLOWforms as desired.) In turn, built in checking processes provide an interactive, semi-automatic
way to create links between converted designs and the data/process resources referenced in those designs.

Depending on the time available, I plan to demonstrate the above capabilities interactively during my talk.

References: