HOUSE OF STRUCTURE

For the first time, a respected computer scientist advocated building a software model before implementing it in code. What a radical concept!

BY WARREN KEUFFEL

Structured Programming. Structured Design. Structured Analysis. Object-oriented Analysis. A lot of structure, but not a lot of understanding. Let's take a look at how the craft of analysis methods has evolved over the past 25 years. And then a look forward at the brave new world of objects.

Why should we look back at all? Haven't the new object-oriented methods discredited the structured movement? Isn't functional decomposition dead, replaced by object browsers? Aren't data-flow diagrams and entity-relationship diagrams as dead as the mainframe? In a word: NO. As I hope will be apparent by the end of this article, those object-oriented gurus who are advocating abandoning the blood, sweat, and tears of the last 25 years are full of more than objects. First, a retrospective look at the development of structured analysis.

Where did structured analysis begin? A good place to draw a line in the sands of time, and say *here* is the 1965 publication of "Programming Considered as a Human Activity" by Edsger Dijkstra in the *Proceedings of the 1965 IFIP Congress.* In this paper, Dijkstra argues that self-modifying code, then considered one of the "virtues of the von Neumann type" of machine is "to a great extent responsible for the lack of clarity" in programs. Here too, Dijkstra first voices his concerns about the value of the GOTO statement: "But I have reasons to ask, whether the goto statement as a remedy is not worse than the defect it is aimed to cure." Dijkstra also first describes "the principle of noninterference"—what will later come to be known as functional decomposition.

But this paper did not create much of a stir among computer professionals. Neither did another paper published the following year by Bohm and Jacopini in the May 1966 *Communications of the ACM,* "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." This was an extremely abstract mathematical paper that few can read without having their eyes glaze over. As Ed Yourdon says, "I have read the paper myself several dozen times during the past 12

Kathryn Thompson

years and honesty requires me to admit that I barely understand it for a period of five minutes after reading the last paragraph." But we should take note of Bohm and Jacopini's work, because it was there that the theoretical foundations for structured programming were first laid. What the authors accomplished in their paper was that they proved that any flow chart can be constructed out of th standard three one-entry/one-ex constructs (iteration, selectio and sequence modules.)

LARRY CONSTANTINE ON STRUCTURED METHODS AND OBJECT ORIENTATION

Larry Constantine is one of those most unusual of men: a Renaissance man equally at home in the world of computing machine problems and in the world of human problems; a genuinely compassionate and likeable individual who transcends the traditional image of the arrogant guru.

Constantine was the architect of structured design, the method that built the bridge between the micro universe of programs and the macro universe of systems. After writing "Structured Design," he left for another career in family therapy. In the preface to his book he wrote his farewell. "No more ghosts in the machine. I can now devote full time to the infinitely more important issues of people in families. Ciao!"

Constantine continues to play a central part in the art and science of software development methods and management. Following are Constantine's views on a variety of issues, taken from a recent UNIX REVIEW *interview.*

ON HIS CAREER.

When the *IBM Systems Journal* paper was being written and published, I was splitting my time between a position as Clinical Professor of Psychiatry in the Tufts University School of Medicine where I was training family therapists at the Center for Training in Family Therapy and part-time consulting and teaching in structured design. For 4½ years, starting in 1968, I was on the faculty of IBM's now defunct Systems Research Institute.

I left because I was more interested in the issues of people getting along with each other and functioning more effectively than in building understanding of how to design better computer programs. I still am.

My decision to return to the field was influenced by many factors, personal and professional, and spread over many months, starting in late 1986, early 1987. By Software Development '88 I was already convinced that object orientation was one of the most important developments in software engineering. In approximately that time frame, I had a series of conversations with George Schussell of Digital Consulting Inc., convincing him that object-orientation was a basic and growing force in the field. Software Development '88 marked my "public" return. P.J. Plauger would have to be credited as a major factor in my return, since he engineered my presence at Software Development '88.

ON THE GENESIS OF STRUCTURED DESIGN.

Virtually all of the content of the paper was my work. [Glenford] Myers did make some later contributions, namely the notion of "informational cohesion" and the modular form of the "information cluster," but the paper was based on ideas I had developed by 1968 and refined during my tenure at SRI. There certainly were many sources of inspiration and ideas in the evolution of my thinking, most of which are pretty well documented in publications. Jim Emory certainly influenced me to apply general systems thinking to software, and his pioneering work on program hierarchy led to developing structure char (once called hierarchy charts). A my work on SD grew out of actu systems development experience: m own, clients, colleagues. The greate influences were my mentors at CEI (later absorbed into Control Data Dave Jasper and Ken Mackenzi Besides the general systems the rists, Miller and the other cognitiv psychologists were the source many ideas, and I believe I was th first to speak of the "magical num ber" (7±2) in our field.

ON THE DEVELOPMENT OF STRUCTURED ANALYSIS.

I had little to do with the evolution SA after laying the groundwork b introducing Martin and Estrin data-flow graphs adapted to analys and design. Most of the developmer of SA took place at Yourdon In There was, of course, far more to S than the 1974 paper dealt wit which was included in courses taught for Yourdon Inc. There we also memos, papers, chapter draft and course notes available to othe staff.

ON OBJECT ORIENTATION.

Since software engineering has nev been a matter of religious doctrine t me, I am still not a convert [to objec orientation]. Object orientation is different paradigm of profound im portance, but it is also just a colle tion of useful ideas for buildin software.

ON THE PATH FROM STRUCTURED DESIGN TO OBJECTS.

I have slowly reached the conclusio that many of the gurus who preac that OO has no relation to SD an function-orientation and require "forgetting everything you learne before about SD and procedur programming" are mostly ignora or misinformed when it comes to SD Often their comparisons contrastin OO to SD exhibit egregiously ba examples of SD that include mi takes only beginners or outside make.

Neither SA nor SD as current practiced have proved to be ver good routes to actually deriving sound OO design, but nearly all the basic principles still apply: prob

With the theoretical foundation of structured programming poured by Bohm and Jacopini, Dijkstra proceeded to lay the joists (and not incidentally ignite a ferocious debate that continued for more than two decades) by writing an innocuous letter to the editors of the *Communications of the ACM*, entitled "Go To Statement Considered Harmful." Published in the March 1968 issue of the CACM, Dijkstra's letter woke up the programming community and essentially divided it into two camps:

lem partitioning, component integrity (cohesion), independence (coupling), etc. All science and engineering builds on what has gone before. It is silly to think that OO in our field would be an exception.

It is not necessary to "unlearn" SD in order to learn OO. This myth has been perpetuated largely by people who lack the training and experience to draw the connections. It is easier for them to say, "Just learn my language and forget everything else you know," than to learn enough about SD to be able to explicate the interrelationships. In truth, it is neuro-physiologically impossible to unlearn anything. Moreover, someone who knows well both SD and OO methods clearly has an upper hand over anyone who knows only one. We do not regard people who are bilingual as handicapped; we do not regard engineers who know only one technology as better off than those versed in several.

There may be some truth to the notion that most practitioners never learned SD very well. The average software developer is, by definition, only average. We must never forget that. In fact, SD was developed specifically to give average developers a better shot at doing what the best designers did. If a professional can't learn and apply the discipline and concepts of SA and SD, why should we think they will do any better with OO?

The job won't be done until the synthesis of OO with SD has been completed (such is the way of all intellectual revolutions, as Hegel had the genius to explain), but I don't know if I'll be around to see it done.

ON GOOD SOFTWARE DESIGN.

Good software design demands knowledge, skill, and discipline. OO does not substitute for either brains or maturity. Because the methodology itself in OO is less well developed at this time, it may even take more maturity and self-discipline to do it well. In some languages, moving to OO multiplies the ways one can screw up and only rigor and care can stave off disaster.

ON EDUCATING PROGRAMMERS.

One of the problems our field faces is that each new generation of programmers often starts from scratch, ignorant of history, ungrounded in basics. New programmers should learn the fundamentals of SP, SA, and SD for the same reason that new physicists have to learn Newtonian mechanics. I'm not talking about the "methodology" here, such as "transform-centered design" or top-down decomposition. I'm talking about the fundamental intellectual content, the principles of structure, partitioning, orderly development, intramodular integrity, intermodular independence, generality, flexibility, extensibility, etc. What we often get is hotshot OO programmers who think that Smalltalk and a browser is a substitute for discipline and fundamental skills, who prototype their way through to spectacular OO kludges that are full of objects but also full of time bombs.

I advocate an "evolutionary approach," if by that is meant an approach that builds on established knowledge and proven principles, that is honest to history, not because that is kind to a generation of SD-trained programmers, however skilled or unskilled they are. I take this approach because that is how science and engineering progress.

I see my potential contribution at this point as two-fold. First, I have focused on investigating and exploring the connections between OO software and conventional function-oriented software in terms of architecture, behavior, and performance. OO is both fundamentally new and different and also related to what has gone before and that is already well understood. I teach OO by building on what people already know and do, stressing connections and relationships, drawing contrasts through detailed comparisons that give equal attention to both sides.

I have tried to approach these issues as a kind "Consumers' Report" of software engineering. I really have no stake in the outcome. My reputation and career are not hanging in the balance and I have no products to sell. I honestly don't care whether OO or SD "wins out" or even whether either one survives. I do care about helping people to do their jobs better, to build better software, and to get more out of it themselves. I am convinced that OO has something significant to offer in this, but no intelligent person should believe that this is the last word on software development methodology. Something else will surely come along. The underlying theory and concepts of function-oriented software are not going to go away after this "paradigmatic revolution," any more than Newtonian physics disappeared after Einstein and Heisenberg.

ON SUCCESSFUL SOFTWARE DEVELOPMENT.

The real determinants of software success are not whether people use objects or functions but how they work, what kind of process they go through, and how they work together. We must remember that some organizations have had long histories of great success with SD and that there are disaster stories in the world of OO. Neither approach is a panacea, neither substitute for good cooperation among intelligent, knowledgeable, and committed people. OO is a good idea because of what it adds, not because SA and SD were a failure. Ed [Yourdon] himself points out that, using SA/SD, some groups routinely deliver 100,000 line systems on time, within budget, and to specifications. OO has yet to prove itself on that scale.

ON ED YOURDON AND YOURDON INC.

Before Ed Yourdon there was neither a field of structured methods nor an industry. More than any other individual, I think Ed is responsible for turning a few scattered ideas into a gigantic enterprise. His genius probably lies in his ability to recognize and promote good people and good ideas. He is responsible for many of the best ideas in SA and for recognizing the good ideas of others. Yourdon Inc. was the spawning ground for an entire generation of the most active and able contributors to the field. He is adept at collecting things into palatable packages. If he has been more an interpreter and conveyor of ideas than an originator and developer, it is because he is so good at it, identifying the best ideas and expressing them in ways that communicate and sell.

those who favored efficiency above all, and those who favored maintainability and readability; those who considered the object code paramount against those who regarded the source code paramount. Dijkstra's argument eventually prevailed, but it was a long, hard sell to convince programmers who had learned to fight for every free byte and every free machine cycle that creating readable, maintainable source code was the best choice for the long run.

Later that year, NATO brought together a group of the world's leading computer scientists in Garmisch, Germany. That conference can arguably be called the genesis of software engineering: It was the first of what later became the annual International Conference on Software Engineering. (The 13th ICSE will be held in May of 1991 in Austin, Texas.) Prior to the Garmisch conference, programming was regarded as an undisciplined art form—hacking, if you will—but the increasing importance of large, reliable computer software systems in military, as well as commercial, activities forced a paradigm shift from programming as a hacking activity, to programming as disciplined engineering activity. There was much resistance to this novel concept, and the Garmisch conference did not cause an immediate change in programmers' habits. But the seed had been sown, and the subflooring had been laid for the structure of software engineering.

Structured programming was gaining momentum. Building on concepts first introduced by Constantine and others, in 1972 David Parnas laid the finish flooring for the House of Structure when he published "A Technique for Software Module Specification with Examples" in the May 1972 *Communications of the ACM*. In this paper, Parnas introduced the concept of pseudocode—structured English-like approximations of the algorithms—to illustrate the central point of his thesis, that "a major contributing factor in the so-called 'software engineering' problem is our lack of techniques for precisely specifying program segments without revealing too much information." Parnas went on to expand on that thought, that developing functional specifications for software was crucial for correct implementation of the user's requirements. This was the first time that anyone had suggested that it was desirable to model a program—create an abstract image of it—prior to writing the actual code. To be sure, flowcharts were in widespread usage, but flowcharts were not sufficiently abstract to satisfy Parnas. Consider the first three points Parnas made.

1. The specification must provide to the intended user *all* of the information that he will need to use the program correctly, *and nothing more*.
2. The specification must provide to the implementer, *all* the information about the intended use that he needs to complete the program, and *no additional information;* in particular, no information about the structure of the calling program should be conveyed.
3. The specification must be sufficiently formal that it can conceivably be machine tested for consistency, completeness (in the sense of defining the outcome of all possible uses) and other desirable properties of a specification.

Here, for the first time, a respected computer scientist—one of the first software engineers—was advocating building an abstract, formal, testable model of a software program before implementing it in code. What a radical idea! What a waste of time! Why fool around with a model when we could be coding? Models are for civil engineers building bridges, not for programmers! (Sound like a familiar argument?)

A well-structured program will have a top-down decomposition of modules. The topmost module should do nothing but control the subordinate modules' behavior.

Up until this time, programmers and software engineers concerned themselves with programs. Systems were simply conglomerations of programs addressing similar needs. The interactions between various programs, if considered at all, were relegated to the appendices of the "Victorian Novel" system specifications.

The May 1974 edition of *IBM Systems Journal* carried a paper by Wayne Stevens, Glen Myers, and Larry Constantine with a simple title, "Structured Design." A simple title, but one that would raise the walls of the software engineering structure, because here several new concepts were introduced to the software engineering community: a graphical notation to describe the behavior of modules, a taxonomy of module types, and the first steps toward a method of analyzing systems.

Stevens, Myers, and Constantine showed how a well-structured program would have a top-down decomposition of modules in which the topmost modules did nothing but control the behavior of the subordinate modules, which in turn would control the behavior of modules that would perform the actual work of the program. Symbols were introduced to show the flow of information and control between modules, thus providing for the first time, a graphical model of the formal, testable specifications described previously by David Parnas. Constantine, in the introduction to his 1979 book *Structured Design*, co-authored with Edward Yourdon, credits Edsger Dijkstra and James Emery with providing the spark that was later fanned into the flame of *Structured Design*. Stevens and Myers were students in Constantine's classes at IBM's Systems Research Institute (SRI); they were responsible for taking Constantine's work and creating one of the first formal publications of the new discipline of structured design.

In addition to the system for graphically describing the relationships among modules known as structure charts, Constantine also devised a taxonomy of module

types within the structure. A full description of those types is beyond the scope of this article, but in a nutshell, he introduced the concepts of *coupling* and *cohesion* as descriptors useful in describing the relative independence of intermodule relationships. A program with a high degree of coupling between modules, for example, would be difficult to maintain because changes to one module would have a high liklihood of creating undesirable side effects in another module. Cohesion, said

METHODS AND CASE

As soon as the first data-flow diagram was drawn, some programmer somewhere undoubtedly tried to automate the process, to eliminate the dozens and hundreds of drawn and redrawn diagrams that represented the analysts' progress toward the system's essence. In other segments of the industry, programmers were attempting to automate code generation from written specifications. In still others, database administrators wrestled with the problem of how to accurately convey the relationships among the different data elements resident in their databases. All of these efforts, and others, evolved into what is today referred to as CASE—computer-aided software engineering. How does one make sense out of this huge market? How does one find a compatible tool? More importantly, how does one move CASE into a corporate culture?

One of the most frequent strategies used in analyzing the CASE marketplace is to divide it into three categories: Upper CASE, Lower CASE, and Integrated CASE.

Upper CASE tools focus on automating the tools used in analysis, such as drawing data-flow diagrams (DFDs) or entity-relationship diagrams (ERDs). Many Upper CASE tools also include support for what might be termed the "abstract" portion of design—automating the nonimplementation-specific details.

A Lower CASE tool is usually tied closely to the hardware and language that it supports. Code generation tools illustrate the properties of Lower CASE tools: They know virtually everything about the target environment, they are very implementation-specific, and they know very little about the analysis decisions that preceeded the physical design.

Integrated CASE tools combine the tools for analysis, design, and implementation under one environment. Theoretically, one can capture the requirements, complete the analysis, create the design, and generate the code with seamless integration between the activities of the different phases of the SDLC. The reality of current CASE technology is somewhat less utopian. There is no easy way to smoothly move from structured analysis to structured design without much human involvement. Since there is no one-to-one correspondence between the the processes on a data-flow diagram and the programs of a completed system, human intervention is required to partition the processes into program functions. Further, data-flow-diagram data stores do not necessarily map one-to-one on database tables, although there is a more direct linkage between the entities of an ERD and the DFD data stores.

Some critics of structured analysis, particularly those with strong object-oriented leanings, feel that the lack of a seamless integration strategy between structured analysis and structured design is sufficient reason to reject structured methods in their entirety. Others claim that the functions of analysis and design should always remain separate; otherwise design considerations begin influencing analysis decisions before it is appropriate to consider them. Many of those who promote structured methods feel that the object-oriented advocates who believe all software development ills can be cured with a good browser, loose sight of the degree to which the structure of existing classes influences the exploration of the application's true requirements.

One of the factors limiting the growth of widely-available Upper CASE tools has been the cost of powerful, high-resolution graphic workstations. Prior to the advent of the microcomputer revolution, graphics workstations were typically limited to six-figure, minicomputer-based, CAD-oriented devices. With the microprocessor and inexpensive high-resolution color displays, the doors were thrown wide open for CASE. Now a RISC workstation running UNIX or a 80486-based personal computer running MS-DOS or OS/2 can be purchased with high-resolution color displays for under $10,000. Upper CASE has finally arrived as a viable tool for the average analyst.

Is the average analyst ready? One of the most important considerations when evaluating CASE tools has nothing to do with the CASE tool itself; the focus should be on the "maturity" of the organization considering CASE. The Software Engineering Institute (SEI) at Pittsburgh's Carnegie Mellon University has proposed a five-level rating scale for evaluating the technological maturity of software development organizations. Most organizations surveyed are still at level one, where chaos is the rule rather than the exception.

Introducing CASE to an organization at Level One is tantamount to providing an infantry platoon with a crewless tank. Unless the soldiers have had previous training, their efforts to coordinate maneuvering and effective firepower will result in much wasted ammunition and time. Similarly, in the CASE world, vendors dislike selling CASE software to organizations where personnel are not trained in its use. Although a few quick sales could be made to unsophisticated organizations, the rapid relegation of the software to the shelf will reflect negatively on the vendor. However, getting training from a vendor may not be the best way to introduce software engineering concepts such as structured analysis.

A vendor's training program will frequently focus on the tool (CASE tools are extremely seductive) to the detriment of learning the underlying methods. Also, returning to the tank analogy, soldiers who have learned the concepts of tank warfare will more readily adapt to unfamiliar equipment—a captured enemy tank, for example—than soldiers who have been taught only how to operate a particular tank. Likewise, in the CASE world, programmers who learn structured analysis in the context of a particular CASE tool will have a tendency to always think of analysis as an operation to be performed with that tool, rather than as a intellectual activity that the tool supports.

Many knowledgeable software developers advocate learning structured methods from independent training organizations prior to purchasing a suite of CASE tools. Then—perhaps after a pilot program conducted with inexpensive MS-DOS-based CASE tools—the organization will be in a better position to more precisely identify its needs and seek out those tools that meet their requirements.