A Plea for Lean Software

Niklaus Wirth
ETH Zürich

Software's girth has surpassed its functionality, largely because hardware advances make this possible. The way to streamline software lies in disciplined methodologies and a return to the essentials.

Memory requirements of today's workstations typically jump substantially—from several to many megabytes—whenever there's a new software release. When demand surpasses capacity, it's time to buy add-on memory. When the system has no more extensibility, it's time to buy a new, more powerful workstation. Do increased performance and functionality keep pace with the increased demand for resources? Mostly the answer is no.

About 25 years ago, an interactive text editor could be designed with as little as 8,000 bytes of storage. (Modern program editors request 100 times that much!) An operating system had to manage with 8,000 bytes, and a compiler had to fit into 32 Kbytes, whereas their modern descendants require megabytes. Has all this inflated software become any faster? On the contrary. Were it not for a thousand times faster hardware, modern software would be utterly unusable.

Enhanced user convenience and functionality supposedly justify the increased size of software, but a closer look reveals these justifications to be shaky. A text editor still performs the reasonably simple task of inserting, deleting, and moving parts of text; a compiler still translates text into executable code; and an operating system still manages memory, disk space, and processor cycles. These basic obligations have not changed with the advent of windows, cut-and-paste strategies, and pop-up menus, nor with the replacement of meaningful command words by pretty icons.

The apparent software explosion is accepted largely because of the staggering progress made by semiconductor technology, which has improved the price/performance ratio to a degree unparalleled by any other branches of technology. For example, from 1978 to 1993 Intel's 80x86 family of processors increased power by a factor of 335, transistor density by a factor of 107, and price by a factor of about 3. The prospects for continuous performance increase are still solid, and there is no sign that software's ravenous appetite will be appeased anytime soon.[1] This development has spawned numerous rules, laws, and corollaries, which are—as is customary in such cases—expressed in general terms; thus they are neither provable nor refutable. With a touch of humor, the following two laws reflect the state of the art admirably well:

- Software expands to fill the available memory. (Parkinson)
- Software is getting slower more rapidly than hardware becomes faster. (Reiser)

Uncontrolled software growth has also been accepted because customers have trouble distinguishing between essential features and those that are just "nice to have." Examples of the latter class: those arbitrarily overlapping windows suggested by the uncritically but widely adopted

0018-9162/95/$4.00 © 1995 IEEE

desktop metaphor; and fancy icons decorating the screen display, such as antique mailboxes and garbage cans that are further enhanced by the visible movement of selected items toward their ultimate destination. These details are cute but not essential, and they have a hidden cost.

CAUSES FOR "FAT SOFTWARE"

Clearly, two contributing factors to the acceptance of ever-growing software are (1) rapidly growing hardware performance and (2) customers' ignorance of features that are essential-versus-nice to have. But perhaps more important than finding reasons for tolerance is questioning the causes: *What drives software toward complexity*?

A primary cause of complexity is that software vendors uncritically adopt almost any feature that users want. Any incompatibility with the original system concept is either ignored or passes unrecognized, which renders the design more complicated and its use more cumbersome. When a system's power is measured by the number of its features, quantity becomes more important than quality. Every new release must offer additional features, even if some don't add functionality.

All features, all the time

Another important reason for software complexity lies in monolithic design, wherein all conceivable features are part of the system's design. Each customer pays for all features but actually uses very few. Ideally, only a basic system with essential facilities would be offered, a system that would lend itself to various extensions. Every customer could then select the extensions genuinely required for a given task.

Increased hardware power has undoubtedly been the primary incentive for vendors to tackle more complex problems, and more complex problems inevitably require more complex solutions. But it is not the *inherent* complexity that should concern us; it is the *self-inflicted* complexity. There are many problems that were solved long ago, but for the same problems we are now offered solutions wrapped in much bulkier software.

Increased complexity results in large part from our recent penchant for friendly user interaction. I've already mentioned windows and icons; color, gray-scales, shadows, pop-ups, pictures, and all kinds of gadgets can easily be added.

To some, complexity equals power

A system's ease of use always should be a primary goal, but that ease should be based on an underlying concept that makes the use almost intuitive. Increasingly, people seem to misinterpret complexity as sophistication, which is baffling—the incomprehensible should cause suspicion rather than admiration.

Possibly this trend results from a mistaken belief that using a somewhat mysterious device confers an aura of power on the user. (What it does confer is a feeling of helplessness, if not impotence.) Therefore, the lure of complexity as a sales incentive is easily understood; complexity promotes customer dependence on the vendor.

It's well known, for example, that major software houses have heavily invested—with success—in customer service, employing hundreds of consultants to answer customer calls around the clock. Much more economical for both producer and consumer, however, would be a product based on a systematic concept—that is, on generally valid rules of inference rather than on tables of rules that are applicable to specific situations only—coupled with systematic documentation and a tutorial. Of course, a customer who pays—in advance—for service contracts is a more stable income source than a customer who has fully mastered a product's use. Industry and academia are probably pursuing very different goals; hence, the emergence of another "law:"

- Customer dependence is more profitable than customer education.

What I find truly baffling are manuals—hundreds of pages long—that accompany software applications, programming languages, and operating systems. Unmistakably, they signal both a contorted design that lacks clear concepts and an intent to hook customers.

This lack of lucid concepts can't alone account for the software explosion. Designing solutions for complicated problems, whether in software or hardware, is a difficult, expensive, and time-consuming process. Hardware's improved price/performance ratio has been achieved more from better technology to duplicate (fabricate) designs than from better design technique mastery. Software, however, is *all* design, and its duplication costs the vendor mere pennies.

Initial designs for sophisticated software applications are invariably complicated, even when developed by competent engineers. Truly good solutions emerge after iterative improvements or after redesigns that exploit new insights, and the most rewarding iterations are those that result in program simplifications. Evolutions of this kind, however, are extremely rare in current software practice—they require time-consuming thought processes that are rarely rewarded. Instead, software inadequacies are typically corrected by quickly conceived *additions* that invariably result in the well-known bulk.

GOOD ENGINEERING IS CHARACTERIZED BY A GRADUAL, STEPWISE REFINEMENT OF PRODUCTS.

Never enough time

Time pressure is probably the foremost reason behind the emergence of bulky software. The time pressure that designers endure discourages careful planning. It also discourages improving acceptable solutions; instead, it encourages quickly conceived software additions and corrections. Time pressure gradually corrupts an engineer's standard of quality and perfection. It has a detrimental effect on people as well as products.

The fact that the vendor whose product is first on the market is generally more successful than the competitor who arrives second, although with a better design, is another detrimental contribution to the computer industry. The tendency to adopt the "first" as the de facto standard is a deplorable phenomenon, based on the same time pressure.

Good engineering is characterized by a gradual, step-

wise refinement of products that yields increased performance under given constraints and with given resources. Software's resource limitations are blithely ignored, however: Rapid increases in processor speed and memory size are commonly believed to compensate for sloppy software design. Meticulous engineering habits do not pay off in the short run, which is one reason why software plays a dubious role among established engineering disciplines.

LANGUAGES AND DESIGN METHODOLOGY

Although software research, which theoretically holds the key to many future technologies, has been heavily supported, its results are seemingly irrelevant to industry. Methodical design, for example, is apparently undesirable because products so developed take too much "time to market." Analytical verification and correctness-proof techniques fare even worse; in addition, these methods require a higher intellectual caliber than that required by the customary "try and fix it" approach. To reduce software complexity by concentrating only on the essentials is a proposal swiftly dismissed as ridiculous in view of customers' love for bells and whistles. When "everything goes" is the modus operandi, methodologies and disciplines are the first casualties.

ABSTRACTION WORKS ONLY WITH LANGUAGES THAT POSTULATE STRICT TYPING OF VARIABLES AND FUNCTIONS; IN THIS RESPECT, C FAILS.

Programming language methodologies are particularly controversial. In the 1970s, it was widely believed that program design must be based on well-structured methods and layers of abstraction with clearly defined specifications. The abstract data type best exemplified this idea and found expression in then-new languages such as Modula-2 and Ada. Today, programmers are abandoning well-structured languages and migrating mostly to C. The C language doesn't even let compilers perform secure type checking, yet this compiler task is by far most helpful to program development in locating early conceptual mistakes. Without type checking, the notion of abstraction in programming languages remains hollow and academic. Abstraction can work only with languages that postulate strict, static typing of every variable and function. In this respect, C fails—it resembles assembler code, where "everything goes."

Reinventing the wheel?

Remarkably enough, the abstract data type has reappeared 25 years after its invention under the heading *object oriented*. This modern term's essence, regarded by many as a panacea, concerns the construction of class (type) hierarchies. Although the older concept hasn't caught on without the newer description "object oriented," programmers recognize the intrinsic strength of the abstract data type and convert to it. To be worthy of the description, an object-oriented language must embody strict, static typing that cannot be breached, whereby programmers can rely on the compiler to identify inconsistencies. Unfortunately, the most popular object-oriented language, C++, is no help here because it has been declared to be upwardly compatible with its ancestor C. Its wide acceptance confirms the following "laws":

- Progress is acceptable only if it's compatible with the current state.
- Adhering to a standard is always safer.

Given this situation, programmers struggle with a language that discourages structured thinking and disciplined program construction (and denies basic compiler support). They also resort to makeshift tools that chiefly add to software's bulk.

What a grim picture; what a pessimist! the reader must be thinking. No hint of computing's bright future, heretofore regarded as a given.

This admittedly somber view is realistic; nonetheless, given the will, there is a way to improve the state of the art.

PROJECT OBERON

Between 1986 and 1989, Jurg Gutknecht and I designed and implemented a new software system—called Oberon—for modern workstations, based on nothing but hardware. Our primary goal was to show that software can be developed with a fraction of the memory capacity and processor power usually required, without sacrificing flexibility, functionality, or user convenience.

The Oberon system has been in use since 1989, serving purposes that include document preparation, software development, and computer-aided design of electronic circuits, among many others. The system includes

- storage management,
- a file system,
- a window display manager,
- a network with servers,
- a compiler, and
- text, graphics, and document editors.

Designed and implemented—from scratch—by two people within three years, Oberon has since been ported to several commercially available workstations and has found many enthusiastic users, particularly since it is freely available.[2]

Our secondary goal was to design a system that could be studied and explained in detail, a system suitable as a software-design case study that could be penetrated top-down and whose design decisions could be stated explicitly. (Indeed, there is a lack of published case studies in software construction, which becomes all the more evident when one is faced with the task of teaching courses.) The result of our efforts is a single book that describes the entire system and contains the source code of all modules.

How is it possible to build a software system with some five man-years of effort and present it in a single book?[3]

Three underlying tenets

First, we *concentrated on the essentials*. We omitted anything that didn't fundamentally contribute to power and flexibility. For example, user interaction in the basic system is confined to textual information—no graphics, pictures, or icons.

Secondly, we wanted to use a truly *object-oriented programming language*, one that was type-safe. This, coupled with our belief that the first tenet must apply even more stringently to the tools than to the system being built, forced us

to design our own language and to construct its compiler as well. It led to Oberon,[4] a language derived from Modula-2 by eliminating less essential features (like subrange and enumeration types) in addition to features known to be unsafe (like type transfer functions and variant records).

Lastly, to be simple, efficient, and useful, we wanted a system to be *flexibly extensible*. This meant that new modules could be added that incorporate new procedures based on calling existing ones. It also meant that new data types could be defined (in new modules), compatible with existing types. We call these *extended types*, and they constitute the only fundamental concept that was added to Modula-2.

Type extension

If, for example, a type *Viewer* is defined in a module called Viewers, then a type *TextViewer* can be defined as an extension of Viewer (typically, in another module that is added to the system). Whatever operations apply to Viewers apply equally to TextViewers, and whatever properties Viewers have, TextViewers have as well.

Extensibility guarantees that modules may later be added to the system without requiring either changes or recompilation. Obviously, type safety is crucial and must cross module boundaries.

Type extension is a typical object-oriented feature. To avoid misleading anthropomorphisms, we prefer to say "TextViewers are compatible with Viewers," rather than "TextViewers inherit from Viewers." We also avoid introducing an entirely new nomenclature for well-known concepts; for example, we stick to the term *type*, avoiding the word *class*; we retain the terms *variable* and *procedure*, avoiding the new terms *instance* and *method*. Clearly, our first tenet—concentrating on essentials—also applies to terminology.

Tale of a data type

An example of a data type will illustrate our strategy of building basic functionality in a core system, with features added according to the system's extensibility.

In the system's core, the data type *Text* is defined as character sequences with the attributes of font, offset, and color. Basic editing operations are provided in a module called *TextFrames*.

An electronic mail module is not included in the core, but can be added when there is a demand. When it is added, the electronic mail module relies on the core system and imports the types *Text* and *TextFrame* displaying texts. This means that normal editing operations can be applied to received e-mail messages. The messages can be modified, copied, and inserted into other texts visible on the screen display by using core operations. The only operations that the e-mail module uniquely provides are receiving, sending, and deleting a message, plus a command to list the mailbox directory.

Operation activation

Another example that illustrates our strategy is the activation of operations. Programs are not executed in Oberon; instead, individual procedures are exported from modules. If a certain module *M* exports a procedure *P*, then *P* can be called (activated) by merely pointing at the string *M.P* appearing in any text visible on the display, that is, by moving the cursor to *M.P* and clicking a mouse button. Such straightforward command activation opens the following possibilities:

1. Frequently used commands are listed in short pieces of text. These are called *tool-texts* and resemble customized menus, although *no special menu software* is required. They are typically displayed in small viewers (windows).
2. By extending the system with a simple graphics editor that provides captions based on Oberon texts, commands can be highlighted and otherwise decorated with boxes and shadings. This results in pop-up and/or pull-down menus, buttons, and icons that are "free" because the basic command activation mechanism is reused.
3. A message received by e-mail can contain commands as well as text. Commands are executed by the recipient's clicking into the message (without copying into a special command window). We use this feature, for example, when announcing new or updated module releases. The message typically contains receive commands followed by lists of module names to be downloaded from the network. The entire process requires only a few mouse clicks.

Keeping it simple

The strategy of keeping the core system simple but extensible rewards the modest user. The Oberon core occupies fewer than 200 Kbytes, including editor and compiler. A computer system based on Oberon needs to be expanded only if large, demanding applications are requested, such as CAD with large memory requirements. If several such applications are used, the system does not require them to be simultaneously loaded. This economy is achieved by the following system properties:

1. *Modules can be loaded on demand.* Demand is signaled either when a command is activated—which is defined in a module not already loaded—or when a module being loaded imports another module not already present. Module loading can also result from data access. For example, when a document that contains graphical elements is accessed by an editor whose graphic package is not open, then this access inherently triggers its loading.
2. *Every module is in memory at most once.* This rule prohibits the creation of linked load files (core images). Typically, linked load files are introduced in operating systems because the process of linking is complicated and time-consuming (sometimes more so than compilation). With Oberon, linking cannot be separated from loading. This is entirely acceptable because the intertwined activities are very fast; they happen automatically the first time a module is referenced.

The price of simplicity

The experienced engineer, realizing that free lunches never are, will now ask, *Where is the price for this economy hidden?* A simplified answer is: in a clear conceptual basis and a well-conceived, appropriate system structure.

If the core—or any other module—is to be successfully extensible, its designer must understand how it will be used. Indeed, the most demanding aspect of system design is its decomposition into modules. Each module is a part with a precisely defined interface that specifies imports and exports.

Each module also encapsulates implementation techniques. All of its procedures must be consistent with respect to handling its exported data types. Precisely defining the right decomposition is difficult and can rarely be achieved without iterations. Iterative (tuning) improvements are of course only possible up to the time of system release.

It is difficult to generalize design rules. If an abstract data type is defined, carefully deliberated basic operations must accompany it, but composite operations should be avoided. It's also safe to say that the long-accepted rule of specification before implementation must be relaxed. Specifications can turn out to be as unsuitable as implementations can turn out to be wrong.

IN CONCLUDING, HERE ARE NINE LESSONS LEARNED from the Oberon project that might be worth considering by anyone embarking on a new software design:

1. The exclusive use of a strongly typed language was the most influential factor in designing this complex system in such a short time. (The manpower was a small fraction of what would typically be expended for comparably sized projects based on other languages.) Static typing (a) lets the compiler pinpoint inconsistencies before program execution; (b) lets the designer change definitions and structures with less danger of negative consequences; and (c) speeds up the improvement process, which could include changes that might not otherwise be considered feasible.
2. The most difficult design task is to find the most appropriate decomposition of the whole into a module hierarchy, minimizing function and code duplications. Oberon is highly supportive in this respect by carrying type checks over module boundaries.
3. Oberon's type extension construct was essential for designing an extensible system wherein new modules added functionality *and* new object classes integrated compatibly with the existing classes or data types. Extensibility is prerequisite to keeping a system streamlined from the outset. It also permits the system to be customized to accommodate specific applications at any time, notably without access to the source code.
4. In an extensible system, the key issue is to identify those primitives that offer the most flexibility for extensions, while avoiding a proliferation of primitives.
5. The belief that complex systems require armies of designers and programmers is wrong. A system that is not understood in its entirety, or at least to a significant degree of detail by a single individual, should probably not be built.
6. Communication problems grow as the size of the design team grows. Whether they are obvious or not, when communication problems predominate, the team and the project are both in deep trouble.
7. Reducing complexity and size must be the goal in every step—in system specification, design, and in detailed programming. A programmer's competence should be judged by the ability to find simple solutions, certainly not by productivity measured in "number of lines ejected per day." Prolific programmers contribute to certain disaster.
8. To gain experience, there is no substitute for one's own programming effort. Organizing a team into managers, designers, programmers, analysts, and users is detrimental. All should participate (with differing degrees of emphasis) in all aspects of development. In particular, everyone—including managers—should also be product users for a time. This last measure is the best guarantee to correct mistakes and perhaps also to eliminate redundancies.
9. Programs should be written and polished until they acquire publication quality. It is infinitely more demanding to design a publishable program than one that "runs." Programs should be written for human readers as well as for computers. If this notion contradicts certain vested interests in the commercial world, it should at least find no resistance in academia.

With Project Oberon we have demonstrated that flexible and powerful systems can be built with substantially fewer resources in less time than usual. The plague of software explosion is not a "law of nature." It is avoidable, and it is the software engineer's task to curtail it. ■

References

1. E. Perratore et al., "Fighting Fatware," *Byte*, Vol. 18, No. 4, Apr. 1993, pp. 98-108.
2. M. Reiser, *The Oberon System*, Addison-Wesley, Reading, Mass., 1991.
3. N. Wirth and J. Gutknecht, *Project Oberon—The Design of an Operating System and Compiler*, Addison-Wesley, Reading, Mass., 1992.
4. M. Reiser and N. Wirth, *Programming in Oberon—Steps Beyond Pascal and Modula*, Addison-Wesley, Reading, Mass., 1992.

Niklaus Wirth *is professor of computer science at the Swiss Federal Institute of Technology (ETH) in Zürich. He designed the programming languages Pascal (1970), Modula (1980), and Oberon (1988), and the workstations Lilith (1980) and Ceres (1986), as well as their operating software.*

Wirth received a PhD from the University of California at Berkeley in 1963. He was awarded the IEEE Emmanuel Piore Prize and the ACM Turing Award (1984). He was named a Computer Pioneer by the IEEE Computer Society and is a Foreign Associate of the National Academy of Engineering.

Readers can contact the author at Institut für Computersysteme, ETH CH-8092 Zürich, Switzerland; e-mail wirth@inf.ethz.ch.