Why software jewels are rare

Article *in* Computer · March 1996

DOI: 10.1109/2.485847 · Source: IEEE Xplore

David Parnas
Middle Road Software

Why Software Jewels Are Rare

David Lorge Parnas
McMaster University

Can "lean" software compete in the marketplace? Can useful software also be elegant? Maybe we can't have it all, but surely there's room for improvement.

For much of my life, I have been a software voyeur, peeking furtively at other people's dirty code. Occasionally, I find a real jewel, a well-structured program written in a consistent style, free of kludges, developed so that each component is simple and organized, and designed so that the product is easy to change. Why, since we have been studying software construction for more than 30 years, don't we find more such jewels? How often is it possible to produce such a jewel of a system? Seldom? Frequently? Always?

The author(s) of elegant systems sometimes write articles to tell us how they wrote that software and to suggest that the rest of us should do what they did. The literature contains many excellent examples.[1-3] Such articles include a lot of good ideas. For me, the T.H.E. system[3] (named for the Technicsche Hogeschool Eindhoven, where the system was built) has served as a source of new ideas and insight for 25 years. And Niklaus Wirth's recent publications[1,2] should be read by every software designer.

Nevertheless, in spite of such helpful articles and many textbooks on software design, software jewels remain rare. Most of the software we see or buy is ugly, unreliable, hard to change, and certainly not something that Wirth or Dijkstra would admire. If published papers contain the secret of success for software, shouldn't we see more jewels?

Although the systems we admire contain useful ideas, these jewels are often produced under conditions that are rare in industry: In particular, their designers are free of the constraints limiting those who must sell their products. In the following sections, I will discuss why the recipes of the masters haven't led to more elegant commercial software and then close with some advice for those who would like to produce better software.

WE WANT SOFTWARE TOOLS MORE THAN SOFTWARE JEWELS

Often, software has grown large and its structure has degraded because designers have repeatedly modified it to integrate new systems or add new features. Everyone, even those who don't want all the added features, must then deal with the complexity resulting from repeated modifications. Software that is repeatedly changed to add the unanticipated features needed to keep pace with the market exhibits a definite "aging" effect and becomes ugly.[4] Wirth suggests that we keep our software lean by sticking to essentials, omitting "bells and whistles."[1] Besides, lean software is likely to be smaller and even faster.

It is difficult to argue with this precept, until we try stating the criteria for distinguishing between essentials and luxuries. Wirth mentions icons and overlapping windows as examples of frills. Icons may appear to be an unnecessary gimmick for some of us, but for others they are an important tool in a life filled with interruptions that force us to switch tasks frequently. Overlapping windows may seem unimportant to Oberon designers, but I find them very useful. (Oberon, a workstation system developed by Wirth and colleagues, is used as an illustration in References 1 and 2.) Offered a jewel or a more useful tool, most customers choose utility. To sell products, you have to add the features the market demands. Not everyone has the luxury of working for a not-for-profit institution.

However, it isn't always necessary to choose between function and elegance. Perhaps I'm too optimistic, but I don't think a designer must omit features to build what Wirth calls "lean software." What *is* necessary is to design the product so that newly added features

- do not eliminate useful capabilities,
- make good use of capabilities already present for other purposes, and
- can be ignored or deleted by people who don't want them.

Giving up certain features to avoid "fat software" is analogous to cutting off a foot because one is overweight: It's neither necessary nor advisable.

The jewels exhibit many of these principles, which, along with others, are described explicitly in the literature.[5-7] Giving up certain features to avoid "fat software" is analogous to cutting off a foot because one is overweight: It's neither necessary nor advisable. Given a choice between tool and jewel, we will choose tool; but with a little more thought, we can often have both. Studying the jewels can show us how.

WE WANT COMPATIBILITY

Software can grow large for many reasons. In "A Plea for Lean Software,"[2] Wirth concentrates on one reason: bad design. A quite different cause of overweight software is the inclusion of features and interfaces necessary to make a new product compatible with earlier ones. When designing a product intended for a large market, software engineers must remember that potential customers will have existing software they want to continue using and existing files they must still be able to process. They will also want the capability to communicate easily with other systems, import documents produced by other systems, operate a wide variety of existing peripheral devices, and so forth. Few users are willing to abandon old files, programs, documents—or even old habits—when switching to a new system. In the real world, designers must add capabilities to their systems that they would not add if they were designing in a vacuum. A designer who doesn't have to worry about sales can pursue a design that allows only one way to perform key functions. Those wishing to enter a very competitive market may find that such elegance will doom their product.

After reading "Gedanken zur Software-Explosion,"[1] I sent an e-mail message to Wirth asking if he had an English version my students could read. He replied, "I can either send you the original in English on paper (edited in readable form), or I can e-mail it as an ASCII text. Let me know what you prefer." In this decade, most of us use computer networks that let us exchange papers in better ways. I can send a LaTex, nroff, or PostScript version of a paper almost anywhere and the recipient will be able to print it. Who would want to use a system that would not allow us to send or receive papers that were prepared using "standard" tools such as these? Providing these capabilities requires either reimplementing the processors for those notations or providing the standard interfaces needed by existing processors. Some may view this as "fat," but others will recognize it as "muscle."

Systems that offer compatibility with other products and earlier systems will never be jewels, but they will be useful.

GOALS VERSUS LIMITATIONS

Performance goals and hardware limitations often interfere with structure. I once belonged to a team that tried to produce a software jewel under tight memory and processor constraints. With the support of the US Navy, our small team tried to redesign the onboard flight software for the A-7E aircraft. We had two constraints: We could not change the hardware, and we could not change the user interface.

Our effort resulted in the publication of many useful design ideas (for example, techniques for describing requirements[8] and for designing interfaces[7] and software architecture[9]). Nevertheless, we failed to reach our goal of producing a running jewel. Inspired by Dijkstra[3] and armed with ideas later published in the literature,[6,10] we thought it would be easy to produce something that performed as well as the unstructured and poorly documented product already in use. We failed because we could not overcome the hardware constraints. For example, the computer had been designed especially for military applications and had a register structure that I found bizarre. Near-optimal register allocation was essential to fitting the program into a very small memory. One of our design goals (inspired by Dijkstra) had been to achieve hardware independence for most of our code. To achieve hardware independence on the specified processor, we needed an effective register allocation algorithm. The previous software for this task had been successful because none of the code was portable and register allocation was done by hand. We never found the necessary register allocation algorithm. The T.H.E. system[3] had been designed as if performance didn't matter and, consequently, its performance didn't satisfy many of its intended users. Commercial success was not one of the goals.

Although today's machines are far better than the one we were using, goals have expanded and competitive pressures often limit the resources available. Few of today's designers are free to ignore performance requirements and hardware limitations. In our attempt to apply Dijkstra's ideas, we discovered that some of them could be refined to reduce performance problems. Several examples in the literature[6,7,11] refine the concept of hierarchy. Unfortunately, applying the more refined ideas requires a lot of time for analysis and backtracking, another luxury not usually available in today's deadline-driven market.

STANDING ON EACH OTHER'S SHOULDERS

The masters have had a chance to learn from others.

The fat and ugly software we use today wasn't written from scratch; it evolved. Software was written, tested, offered to users, and then changed in response to their requests. Programs were modified to offer new (and more general, convenient, or intuitive) features. If the designers of fat software were allowed to start over, designing what they would have designed if they had known what was coming, their products would look very different. Given the opportunity to discard all the old ways of doing things and to just do it right the first time, they would probably produce lean and efficient software. Indeed, we'd all do better if we could start with all the knowledge we will have later when a product is mature. Unfortunately, commercial designers don't have that chance very often.

Designers of software jewels often had the advantage of being able to learn from others' mistakes. For example, the designers of Oberon gained by watching other teams design similar systems. Theirs is not the first workstation offering storage management, a file system, a window display manager, a network with servers, a compiler, and editors. Wirth and his team, for example, were closely connected with Xerox PARC. Similarly, the designers of T.H.E.[3] knew about developments on a variety of other operating systems. Those working on the programs that control the US telephone system have estimated that they could replace 25 million lines of code with a program that is a small fraction of that size if they could start over. When Wirth asks, rhetorically, how Oberon could be so small, he doesn't give the whole answer. The Oberon design team obviously learned a great deal from the mistakes of others, and those others have not had a chance to return the compliment.

There is a positive lesson in this for those who do have the opportunity to start a new project. Time spent studying previous efforts and identifying the reasons for their poor structure is likely to pay off in a far better, easier to maintain, product.

REINVENTING THE WHEEL

One of the weaknesses of technological society is that we sometimes place far too much emphasis on originality. Creativity and originality are obviously valuable wherever there is room for improvement, and they are essential when dealing with problems for which we have no adequate solution.

Nevertheless, we have an unfortunate tendency to value creativity as an end in itself and use it as an excuse for ignorance. I have known both researchers and developers who refused to look at previous work because they wanted to use their own ideas. Managers often do not allow their designers time to study the way things have been done in the past. It seems obvious that we should use our own ideas only if they are better than previous ones. Successful innovators usually know previous work and have managed to understand the fundamental weaknesses in earlier approaches. Too many software products show evidence of "ignorant originality." They make the same mistakes others made before them and ignore solutions that others have found.

Wirth appears critical of the purveyors of techniques that use the buzzword "object-oriented" for having reinvented the ideas behind the older concept of abstract data type.[1,2] However, many would argue that abstract data type itself was a reinvention (refinement) of ideas that appeared in even earlier work (for example, References 3, 10, and 12). Through his best known language, Pascal, Wirth is often given credit for "inventing" ideas that I first saw in Algol-60 and other early languages. Nobody criticizes Pascal's inventors for having reused good ideas; it would have been foolish and irresponsible not to. We must not forget that the wheel is reinvented so often because it is a very good idea; I've learned to worry more about the soundness of ideas that were invented only once.

Sometimes the introduction of new words for old ideas blocks the old literature from view. Newcomers, entranced by OO terminology, don't even read older papers on software design in which some of the "new" ideas are nicely described and illustrated. Everyone who likes OO ideas should read "The Structure of the T.H.E. Multiprogramming System,"[3] which describes an object-oriented design without ever using the word *object*.

DESIGN VERSUS LANGUAGE

Sometimes new languages are used in the design of jewels, and authors may attribute a product's success to the use of a particular language or type of language. Here, I have grave doubts. I have lost count of the number of languages that have been introduced to me as the solution to the software problems that everyone experiences. First, I was told to use Fortran instead of an assembler language. Later, others advocated Algol-60 and its derivatives as the cure to the ugly software resulting from Fortran. Of course, NPL, later known as PL/I, was going to provide an even better solution. The list goes on. Wirth promotes Oberon[2] while hundreds of people are telling me that an object-oriented language must be used to get clean software. I no longer believe any such claims. The issue is design, not programming language.

The wheel is reinvented so often because it is a very good idea; I've learned to worry more about the soundness of ideas that were invented only once.

Wirth is best known for his work as a designer of languages, so it is not surprising that he views the problems of software design as a question of language.[1,2] Computer science's greatest contributions have been in the area of language design, and designing a new language is a reflex for many trained in that discipline. However, my experience does not support the view that the programming language used determines the quality of the software. I have seen beautiful, lean software written using only an assembler (Dijkstra offers an example[3]), good software written in Fortran, and even good software written in C. I have also seen programs in which each of these tools was used badly.

In an ideal world, today's most popular languages would not be my first choice as program construction tools, and I think Wirth's criticisms of C are quite valid. However, product designers can rarely choose what language to use. They are required to interface with legacy code and use a language known by many programmers. The option of designing a new language for each new project is rare in a commercial environment. Focusing on the programming language is a red herring that will distract us from real solutions to the problem of poor software.

The jewels I've found owe their elegance to

- the use of good decomposition principles, as discussed in Reference 12;
- the use of good hierarchical structures, as discussed in References 3, 5, 6, and 11; and
- the design of interfaces, as discussed in References 5 and 7.

The design principles presented in these papers can be applied in any language.

We should not ignore the fact that most modern languages have inherent disadvantages. A language that supports a certain approach to software design often compels us to use a particular implementation of a design principle, one that may be inappropriate for the task at hand. For example, many languages that support modules, abstract data types, and object classes require the use of subroutines where macro expansion might be a better choice. Moreover, languages that prevent programming errors, a goal advanced by some inveterate language designers, are as feasible as knives that can cut meat but not hands. We need sharp tools to do good work.

Languages that prevent programming errors, a goal advanced by some inveterate language designers, are as feasible as knives that can cut meat but not hands.

THERE IS MUCH TO LEARN FROM JEWEL-LIKE SYSTEMS. We can and *must* learn to write lean software and systems like Oberon and T.H.E., systems that provide important lessons. We can apply those lessons even if we write in C or assembler, and we can use the good design principles to write better software even if commercial constraints mean that the product can't be as small and elegant as the jewels we would all like to manufacture. The most important lesson is "up-front investment." In each of the jewels I've seen, the designers had obviously spent a lot of time thinking about the structure of their system before writing code. The system structure could be accurately described and documented without reference to the code. Programs were not just written; they had been planned, often in some pseudocode or a language other than the actual programming language. In contrast, the worst software I've seen was written in "stream of execution order" without a design having been produced (and reviewed) in advance.

My engineering teachers laid down some basic rules:

1. Design before implementing.
2. Document your design.
3. Review and analyze the documented design.
4. Review implementation for consistency with the design.

These rules apply to software at least as much as they do to circuits or machines. ∎

References

1. N. Wirth, "Gedanken zur Software-Explosion," *Informatik Spektrum*, Band 17, Heft 1, Feb. 1994.
2. N. Wirth, "A Plea for Lean Software," *Computer*, Vol. 28, No. 2, Feb. 1995, pp. 64-68.
3. E.W. Dijkstra, "The Structure of the THE Multiprogramming System," *Comm. ACM*, Vol. 11, No. 5, May 1968, pp. 341-346.
4. D.L. Parnas, "Software Aging," *Proc. 16th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 5855, 1994, pp. 279-287.
5. D.L. Parnas and D.P. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Comm. ACM*, Vol. 18, No. 7, July 1975, pp. 401-408.
6. D.L. Parnas, "Designing Software for Extension and Contraction," *Proc. Third Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 187, 1978, pp. 264-277; also published in *IEEE Trans. Software Eng.*, Mar. 1979, pp. 128-138.
7. K.H. Britton, R.A. Parker, and D.L. Parnas, "A Procedure for Designing Abstract Interfaces for Device Interface Modules," *Proc. Fifth Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 332, 1981, pp. 195-204.
8. K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans. Software Eng.*, Vol. SE-6, Jan. 1980, pp. 2-13.
9. D.L. Parnas, P.C. Clements, and D.M. Weiss, "The Modular Structure of Complex Systems," *IEEE Trans. Software Eng.* (special issue on the Seventh International Conference on Software Engineering), Vol. SE-11, No. 3, Mar. 1985, pp. 259-266.
10. D.L. Parnas, "Information Distribution Aspects of Design Methodology," *Proc. IFIP Congress 1971*, North-Holland, 1972, pp. 26-30.
11. D.L. Parnas, "On a 'Buzzword': Hierarchical Structure," *Proc. IFIP Congress 74*, North-Holland, 1974.
12. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, Dec. 1972, pp. 1,053-1,058.

***David Lorge Parnas** holds the NSERC/Bell Industrial Research Chair in Software Engineering in the Department of Electrical and Computer Engineering at McMaster University in Hamilton, Ontario. In addition to the usual academic positions, Parnas has worked as a consultant for Philips, the US Navy, IBM, and the Atomic Energy Control Board of Canada. The author of more than 180 papers and reports, he is interested in most aspects of computer system design and seeks to find a "middle road" between theory and practice. He received a PhD in electrical engineering from Carnegie Mellon and an honorary doctorate from the ETH in Zurich. The winner of the ACM Best Paper Award in 1979, as well as two Most Influential Paper Awards from the International Conference on Software Engineering, he was also the first winner of the Norbert Wiener Award for Professional and Social Responsibility (awarded by Computing Professionals for Social Responsibility). Parnas is a fellow of the Royal Society of Canada and of the ACM, and a senior member of the IEEE.*

Parnas can be contacted at the Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario, Canada, L8S 4K1.

Ted Lewis, formerly Computer's *Cybersquare area editor, coordinated the review of this article and recommended it for publication. His e-mail address is lewis@cs.nps.navy.mil.*