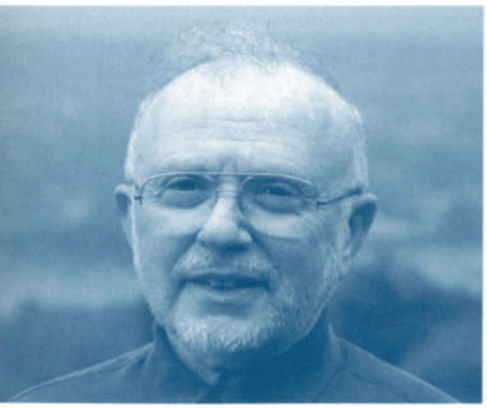
On the Criteria to be Used in Decomposing Systems into Modules

David L. Parnas
Professor of Software Engineering
McMaster University
parnas@qusunt.cas.mcmaster.ca



Ph.D. in Electrical Engineering, Carnegie Mellon University

Advisor, Philips, IBM, United States Naval Research Laboratory, Atomic Energy Control Board of Canada

ACM SIGSOFT Outstanding Research Award, ACM Best Paper Award, ICSE Most Influential Paper Award

Honorary degree from the ETH Zurich and from the University of Louvain in Belgium

Fellow of ACM and Royal Society of Canada

Major contributions: information hiding principle, modularization, module specification

Current interests: safety-critical real-time software, computer system design

M. Broy et al. (eds.), *Software Pioneers*

M. Broy, E. Denert (Eds.): Software Pioneers

David L. Parnas

The Secret History of Information Hiding

The concept of "information-hiding" as a software design principle is widely accepted in academic circles. Many successful designs can be seen as successful applications of abstraction or information hiding. On the other hand, most industrial software developers do not apply the idea and many consider it unrealistic. This paper describes how the idea developed, discusses difficulties in its application, and speculates on why it has not been completely successful.

1. A Dutch Adventure

The question of how to divide systems into modules was not on my mind in 1969 when I took a one year leave-of-absence from my academic position to work with a start-up, Philips Computer Industry, in the Netherlands. I was interested in the use of simulation in software development and wanted to see if my ideas could be made to work in industry. I never found out! Instead, I came to understand that software development in industry was at a far more primitive level than my academic experience would have suggested. The ideas that I had gone to Holland to explore were obviously unrealistic.

I had heard and read many arguments in favour of modular programming and those arguments seemed *obviously* correct. The message was clear; software should be designed as a set of independently changeable components. With a background in electrical engineering, I could not even imagine anyone trying to build a complex product except as an assembly of smaller components or modules. Moreover, I was used to having modules with well-designed, relatively simple, interfaces. I was puzzled by the complexity and arbitrary nature of the interfaces that were being discussed.

At that time, in electrical engineering, the modules or components were presented to us; we learned about how to assemble them into larger circuits. We never had occasion to debate the nature of a component. It was exposure to actual industrial software development that made me realise that in software, the nature of a component, or module, is not obvious. In fact people are rarely able to define those terms unless the tools define them. In academia and papers I had seen a few well-organised systems (Dijkstra's THE system for example [5]), but as I became familiar with commercial software, I began to realise that it takes both deep understanding and a lot of effort to keep software simple and easy to understand.

2. Phrases Like "Clean Design" or "Beautiful" Are Not Meaningful

Philips gave me the opportunity to watch the design of a new product. As it developed, I became increasingly uneasy and started to confide my concerns to my supervisor. I repeatedly told him that the designs were not "clean" and that they should be "beautiful" in the way that Dijkstra's designs were beautiful. He did not find this to be a useful comment. He reminded me that we were engineers, not artists. In his mind I was simply expressing my taste and he wanted to know what criterion I was using in making my judgement. I was unable to tell him! Because I could not explain what principles were being violated, my comments were simply ignored and the work went on. Sensing that there must be a way to explain and provide motivation for my "taste", I began to try to find a principle that would distinguish good designs from bad ones. Engineers are very pragmatic; they may admire beauty, but they seek utility. I tried to find an explanation of why "beauty" was useful.

3. The Napkin of Doom

The first glimmer of understanding came during lunch. Two members of the software team began to discuss work. Jan, who was building a compiler, asked Johan, the database expert, "What does it mean to open a file anyway?" Johan and I looked at him strangely; surely he knew what "open" did. You could not do anything to a file until you had opened it. Johan said that, and Jan replied, "No, what does it *really* mean? I have to implement the open command and I need to know what code to generate." Johan grabbed a napkin from the dispenser on the table and began to draw a dia-

gram of a control block. As he drew, he explained the meaning of each field and indicated where specific addresses and control bits were located.

I felt that drawing the picture was a mistake and tried to stop them. I told them that they would be sorry if they continued. When they asked why, I replied only that they should not have to exchange such information. They insisted that the compiler writers needed that information and tried to get rid of me because I was interfering with their progress. Because I could neither explain why I thought they were making a mistake, nor offer them a better way to solve the problem, I became uncharacteristically silent and watched as the discussion was completed. The napkin was taken upstairs, two holes were punched in it, and it was inserted in a binder. Later, other members of the compiler team made copies of the napkin and put them in their own files.

Months later, I had evidence that I had been right. Johan forgot about the napkin. When it was necessary to change the control block format, the napkin was not updated. Jan's team continued to work as if the control block had not changed. When the two products were integrated, they did not work properly and it took some time to figure out why and even longer to correct the compiler. Each change in the control block format triggered a change in both the compiler and the data base system; this undesired coupling often consumed valuable time.

At this point it was clear to me that I had been right but I still was bothered by what Jan and Johan had said. They too were correct. The compiler group did need some information that allowed the necessary code to be generated. I could not tell them what they could have done instead of exchanging the details on the napkin.

4. Information Distribution in Software

As my time in Europe came to an end, I had some of the concept but not all of it. My first paper on "information hiding" was presented at the IFIP conference in 1971 [8]. Although such conferences are intended to bring together people from all over the world, sessions are not widely attended nor are the proceedings well read. Nonetheless, the paper did explain that it was information distribution that made systems "dirty" by establishing almost invisible connections between supposedly independent modules.

5. Teaching an Early Software Engineering Course

When I returned to teaching, I was told that, since I was an engineer[1] and now had industrial software development experience, I should teach the new "software engineering" course. Nobody could tell me what the content of that course should be or how it should differ from an advanced programming course. I was told to teach whatever I thought was important.[2] After some thought, it became clear to me that information distribution, and

1 Actually, I was not an engineer but had engineering degrees. At that time I had not applied for a an engineering license as I was not practising engineering. Like most scientists, the department head did not understand the difference.

2 I fear that the same thing happens in many computer science departments today.

how to avoid it, had to be a big part of that course. I decided to do this by means of a project with limited information distribution and demonstrate the benefits of a "clean" design [11]. I also found a definition of "module" that was independent of any tools, a work assignment for an individual or a group.

To teach this first course, I decided to take the concept of modularity seriously. I designed a project that was divided into five modules and divided the class into five groups. Every individual in a group would have the same work assignment (module) but would use a different method to complete it. With four members per group, we hoped to get four implementations of each module. The implementations of a module were required to be interchangeable so that (if everything worked) we would end up with 1024 executable combinations. Twenty-five randomly selected combinations were tested and (eventually) ran.

The results of this experiment seemed significant to me because the construction of interchangeable software modules seemed unachievable by others at the time. In fact, even today I see how difficult it can be to install an upgrade in the software we all use. Exchanging one implementation of a module for another rarely goes smoothly. There are almost always complex interfaces and many tiny details that keep the "help lines" busy whenever new versions of minor modules are installed.

6. What Johan *Should* Have Said to Jan

In conducting the experiment as an educational exercise, I had finally found the answer to the issue raised a year earlier by Jan and Johan. Because each module had four really different implementations, my students could not exchange the kind of detailed information that Jan and Johan had discussed. Instead I designed an abstract interface, one that all versions of a module could have in common. We treated the description of the interface as a specification, i.e. all implementations were required to satisfy that description. Because we did not tell the students which combinations would be tested, they could not use any additional information. In effect, the implementation details were kept a secret.

The paper that described the experiment and its outcome was published at a major conference but has not been widely read or cited [12].

7. Two Journal Articles on Information Hiding

There were two journal articles that followed and these are widely cited though sometimes it appears that those who cite them have not read them and many who have read them refer to them without citing them. The better of the two articles [10] was first rejected by the journal. The reviewer stated, "Obviously Parnas does not know what he is talking about because

nobody does it that way." I objected by pointing out that one should not reject an idea claiming to be new simply because it was new. The editor then published the article without major change.

7.1 Language - Often the Wrong Issue

Because it had taken so long to find a solution to the dilemma posed by Jan and Johan, my first thought was that it was the specification notation that was significant. Consequently, the first *Communications of the ACM* article [9] discussed the specification technique and illustrated the idea by publishing the specifications with many typographic errors introduced in the (manual) typesetting process.

It took several discussions with people who had read the article before I realised the significance of the (obvious) fact that the specification method would not work with an arbitrary set of modules. Most important, it would not have worked with conventionally designed modules. The secret of the experiment's success was not the specification notation but the decomposition.

Over the years I have proposed and studied more general approaches to module specification [1, 18]. Many others have also looked at this problem. Slowly I have come to the realisation that standard mathematical notation is better than most languages that computer scientists can invent. I have learned that it is more important to learn how to use existing mathematical concepts and notation than to invent new, ad hoc, notations.

7.2 What Really Matters Is Design

It was the fact that the modularisation limited the amount of information that had to be shared by module implementers that made the effort a success. After some reflection I realised that I had found the answer to the question I had been asked by my manager at Philips. The difference between the designs that I found "beautiful" and those that I found ugly was that the beautiful designs encapsulated changeable or arbitrary information. Because the interfaces contained only "solid" or lasting information, they appeared simple and could be called beautiful. To keep the interfaces "beautiful", one needs to structure the system so that all arbitrary or changeable or very detailed information is hidden.

The second article explained this although it never used the phrase "information hiding". It is possible to formalise the concept of information hiding using Shannon's theory of information. Interfaces are a set of assumptions that a user may make about the module. The less likely these assumptions are to change (i.e. the more arbitrary the assumptions are), the more "beautiful" they appear. Thus, interfaces that contain less information in Shannon's sense are more "beautiful". Unfortunately, this formalisation is not very useful. Applications of Shannon's theories require us to estimate the probability of the "signals" and we never have such data.

7.3 From Parnas-Modules to Information-Hiding Modules

The first large group to build on the information hiding idea was SRI International. They introduced the HDM or Hierarchical Development Method described in [19–22]. They referred to the modules as "Parnas-Modules". I requested that they switch to more descriptive terminology, "information-hiding" modules, and the term "information-hiding" is the one that has been accepted. This term is now often used in books and articles without either citation or explanation.

8. Information Hiding Is Harder Than It Looks

Information hiding, like dieting, is something that is more easily described than done. The principle is simple and is easily explained in terms of established mathematics and information theory. When it is applied properly it is usually successful. Abstractions such as Unix pipes, postscript or X-windows achieve the goal of allowing several implementations of the same interface. However, although the principle was published 30 years ago, today's software is full of places where information is not hidden, interfaces are complex, and changes are unreasonably difficult. In part, this can be attributed to a lack of educational standards for professional software developers. It is quite easy to get a job developing software with little or no exposure to the concept. Moreover, if one's manager and co-workers do not know the concept, they will not be supportive of the use of information hiding by someone who does understand it. They will argue that, "Nobody does it that way" and that they cannot afford to experiment with new ideas.

In addition to the fact that software engineering educational standards are much lower than those for other engineering fields, I have observed other reasons for failing to apply information hiding.

8.1 The Flow-Chart Instinct

My original experiments using the KWIC index revealed that most designers design software by thinking about the flow of control. Often they begin with a flowchart. The boxes in the flowcharts are then called modules. Unfortunately modules developed in this way exchange large amounts of information in complex data structures and have "ugly" interfaces. Flowchart-based decomposition almost always violates the information-hiding principle. The habit of beginning with a flowchart dies hard. I can teach the principle, warn against basing a modularisation on a flowchart, and watch my students do exactly that one hour later.

8.2 Planning for Change Seems Like Too Much Planning

Proper application of the information-hiding principle requires a great deal of analysis. One must study the class of applications and determine which aspects are unlikely to change while identifying changeable decisions.

Simple intuition is not adequate for this task. When designing software, I have identified certain facts as likely to change only to learn later that there were fundamental reasons why they would not change. Even worse, one might assume that certain facts will never change (e.g. the length of a local phone number or the length of a Postleitzahl). Differentiating what will not change from things that can be mentioned in interface descriptions takes a lot of time. Often a developer is much more concerned with reducing development cost, or time-to-market, than with reducing maintenance cost. This is especially true when the development group will not be responsible for maintenance. Why should a manager spend money from his/her own budget to save some other manager money in the future?

8.3 Bad Models

Engineers in any field are not always innovative. They often take an earlier product as a model making as few changes as possible. Most of our existing software products were designed without use of information hiding and serve as bad models when used in this way. Innovative information-hiding designs are often rejected because they are not conventional.

8.4 Extending Bad Software

Given the large base of installed software, most new products are developed as extensions to that base. If the old software has complex interfaces, the extended software will have the same problem. There are ways around this but they are not frequently applied. Usually, if information hiding has not been applied in the base software, future extensions will not apply it either.

8.5 Assumptions Often Go Unnoticed

My original example of information hiding was the simple KWIC index program described in [10]. This program is still used as an example of the principle. Only once has anyone noticed that it contains a flaw caused by a failure to hide an important assumption. Every module in the system was written with the knowledge that the data comprised strings of strings. This led to a very inefficient sorting algorithm because comparing two strings, an operation that would be repeated many times, is relatively slow. Considerable speed-up could be obtained if the words were sorted once and replaced by a set of integers with the property that if two words are alphabetically ordered, the integers representing them will have the same order. Sorting strings of integers can be done much more quickly than sorting strings of strings. The module interfaces described in [9] do not allow this simple improvement to be confined to one module.

My mistake illustrates how easy it is to distribute information unnecessarily. This is a very common error when people attempt to use the information-hiding principle. While the basic idea is to hide information that is likely to change, one can often profit by hiding other information as well because it allows the re-use of algorithms or the use of more efficient algorithms.

8.6 Reflecting the System Environment in the Software Structure

Several software design "experts" have suggested that one should reflect exciting business structures and file structures in the structure of the software. In my experience, this speeds up the software development process (by making decisions quickly) but leads to software that is a burden on its owners should they try to update their data structures or change their organisation. Reflecting changeable facts in software structure is a violation of the information-hiding principle.

In determining requirements it is very important to know about the environment but it is rarely the right "move" to reflect that environment in the program structure.

8.7 "Oh, Too Bad We Changed It"

Even when information hiding has been successfully applied, it often fails in the long run because of inadequate documentation. I have seen firms where the design criteria were not explained to the maintenance programmers. In these cases, the structure of the system quickly deteriorated because good design decisions, which made no sense to the maintainers, were reversed. In one case, I pointed out the advantages of a particular software structure only to see a glimmer of understanding in the eyes of my listener as he said, "Oh, too bad we changed it."

9. A Hierarchical Modular Structure for Complex Systems

My original articles used very small systems as examples. Systems with 5, 10 or 15 modules can be comprehended by most programmers. However, when applying the principle to a problem with hundreds of independently changeable design decisions, interface details and requirements, we recognised that it would be difficult for a maintenance programmer to have the overview needed to find relevant parts of the code when a change was needed. To overcome this, we organised the modules into a hierarchy. Each of the top-level modules had a clearly defined secret, and was divided into smaller modules using the same principle. This resulted in a tree structure. The programmer can find the relevant modules by selecting a module at each level and moving on to a more detailed decomposition. The structure and document are described in detail in [16].

10. There Are No "Levels of Abstraction"!

Some authors and teachers have used the phrase "levels of abstraction" when talking about systems in which abstraction or information hiding has been applied. Although the terminology seems innocent it is not clearly defined and can lead to serious design errors. The term "levels" can be used properly only if one has defined a relation that is loop free [13]. None of

the authors who use this phrase have defined the relation, "more abstract than", which is needed to understand the levels. What would such a relation really mean? Is a module that hides the representation of angles more (or less) abstract than a program that hides the details of a peripheral device? If not, should both modules be on the same level? If so, what would that mean? Using the phrase "levels of abstraction" without a useful definition of the defining relation is a sign of confusion or carelessness.

Often the phrase "levels of abstraction" is used by those who do not clearly distinguish between modules (collections of programs) and the programs themselves. It has been shown useful to arrange programs in a hierarchical structure based on the relation "uses" where A uses B means that B must be present when A runs [5]. However, as shown in [15] it is not usually right to put all programs in a module at the same level in the "uses" hierarchy. Some parts of a module may use programs at a higher level than other programs in that module. An illustration of this can be found in [15].

11. Newer Ideas: Abstract Data Types, O-O, Components

The concept of information hiding is the basis of three more recent concepts. The following paragraphs briefly describe these.

11.1 Abstract Data Types

The notion of abstract data types corrected an important oversight in my original treatment of information hiding. In the examples that motivated my work, there was only one copy of each hidden data structure. Others (most notably Barbara Liskov and John Guttag) recognised that we might want many copies of such structures and that this would make it possible to implement new data types in which the implementation (representation plus algorithms) was hidden. Being able to use variables of these new, user-defined, abstract data types in exactly the way as we use variables of built-in data types is obviously a good idea. Unfortunately, I have never seen a language that achieved this.

11.2 Object-Oriented Languages

A side effect of the application of information hiding is the creation of new objects that store data. Objects can be found in Dijkstra's THE system [5] and many subsequent examples. Information-hiding objects can be created in any language; my first experiments used primitive versions of FORTRAN. Some subsequent languages such as Pascal actually made things a little harder by introducing restrictions on where variables could be declared making it necessary to make hidden data global. It was discovered that a family of languages initially designed for simulation purposes, SIMULA [3] and SIMULA 67 [4], proved to be convenient for this type of programming. The concept of process was introduced to model ongoing activities in the system being simulated. Each process could store data and there were ways for other processes to send "messages" that altered this data. Thus,

the features of SIMULA languages were well suited to writing software that hid data and algorithms. Similar ideas were later found in Smalltalk [6]. In both of these languages, the concept of module (class) and process were linked, i.e. modules could operate in parallel. This introduces unnecessary overhead in many situations.

More recent languages have added new types of features (known as inheritance) designed to make it possible to share representations between objects. Often, these features are misused and result in a violation of information hiding and programs that are hard to change.

The most negative effect of the development of O-O languages has been to distract programmers from design principles. Many seem to believe that if they write their program in an O-O language, they will write O-O programs. Nothing could be further from the truth.

11.3 Component-Oriented Design

The old problems and dreams are still with us. Only the words are new.

12. Future Research: Standard Information – Hiding Designs

Thirty years after I was able to formulate the principle of information hiding, it is clear to me that this is the right principle. Further, I see few open questions about the principle. However, as outlined in Section 8, information hiding is not easy. As discussed in [2], [14] and [16], designing such modules requires a lot of research. I would hope that in the next decade we will see academic research turning in this (useful) direction and journals that present module structures and module interfaces for specific devices, data structures, requirements classes and algorithms. Examples of reusable module designs can be found in [17].

References

[1] Bartussek, W., Parnas, D.L., "Using Assertions About Traces to Write Abstract Specifications for Software Modules", UNC Report No. TR77-012, December 1977; Lecture Notes in Computer Science (65), *Information Systems Methodology, Proceedings ICS*, Venice, 1978, Springer Verlag, pp. 211-236.
Also in *Software Specification Techniques* edited by N. Gehani & A.D. McGettrick, AT&T Bell Telephone Laboratories, 1985, pp. 111-130 (QA 76.6 S6437).
Reprinted as Chapter 1 in item .

[2] Britton, K.H., Parker, R.A., Parnas, D.L., "A Procedure for Designing Abstract Interfaces for Device Interface Modules", *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[3] Dahl, Ol.-J., Nygaard, K., "SIMULA - An Algol-based Simulation Language", *Communications of the ACM*, vol. 9, no. 9, pp. 671-678, 1966.

[4] Dahl, O-J., et al., "The Simula 67 Common Base Language", 1970, Norwegian Computing Centre, publication S22.

[5] Dijkstra, E.W., "The Structure of the THE Multiprogramming System", *Communications of the ACM*, vol. 11, no. 5, May 1968.

[6] Goldberg, A., "Smalltalk-80: The Interactive Programming Environment", Addison-Wesley, 1984.

[7] Hoffman, D.M., Weiss, D.M. (eds.), *"Software Fundamentals: Collected Papers by David L. Parnas"*, Addison-Wesley, 2001, ISBN 0-201-70369-6.

[8] Parnas, D.L., "Information Distributions Aspects of Design Methodology", *Proceedings of IFIP Congress '71*, 1971, Booklet TA-3, pp. 26-30.

[9] Parnas, D.L., "A Technique for Software Module Specification with Examples", *Communications of the ACM*, vol. 15, no. 5, pp. 330-336, 1972;
Republished in Yourdon, E.N. (ed.), *Writings of the Revolution*, Yourdon Press, 1982, pp. 5-18:
in Gehani, N., McGettrick, A.D. (eds.), *Software Specification Techniques*, AT&T Bell Telephone Laboratories (QA 76.7 S6437), 1985, pp. 75-88 .

[10] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972;.
Republished in Laplante P. (ed.), *Great Papers in Computer Science*, West Publishing Co, Minneapolis/St. Paul, 1996, pp. 433-441.
Translated into Japanese - BIT, vol. 14, no. 3, pp. 54-60, 1982.
Republished in Yourdon, E.N. (ed.), *Classics in Software Engineering*, Yourdon Press, 1979, pp. 141-150.
Reprinted as Chapter 7 in item [7].

[11] Parnas, D.L., "A Course on Software Engineering Techniques", in *Proceedings of the ACM SIGCSE*, Second Technical Symposium, 24-25 March 1972, pp. 1-6.

[12] Parnas, D.L., "Some Conclusions from an Experiment in Software Engineering Techniques", *Proceedings of the 1972 FJCC*, 41, Part I, pp. 325-330.

[13] Parnas, D.L., "On a 'Buzzword': Hierarchical Structure", *IFIP Congress '74*, North-Holland, 1974, pp. 336-339.
Reprinted as Chapter 8 in item [7].

[14] Parnas, D.L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems", NRL Report No. 8047, June 1977;
Reprinted in Infotech State of the Art Report, Structured System Development, Infotech International, 1979.

[15] Parnas D.L., "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering*, Vol. SE-5 No. 2, pp. 128-138, March 1979;
In *Proceedings of the Third International Conference on Software Engineering*, May 1978, pp. 264-277.

[16] Parnas, D.L., Clements, P.C., Weiss, D.M., "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering*, Vol. SE-11 No. 3, pp. 259-266, 1985;
In *Proceedings of 7th International Conference on Software Engineering*, March 1984, 408-417;
Reprinted in Peterson, G.E. (ed.), IEEE Tutorial: "Object-Oriented Computing", Vol. 2: Implementations, IEEE Computer Society Press, IEEE Catalog Number EH0264-2, ISBN 0-8186-4822-8, 1987, pp. 162-169;
Reprinted as Chapter 16 in item [7].

[17] Parker, R.A., Heninger, K.L., Parnas, D.L., Shore, J.E., "Abstract Interface Specifications for the A-7E Device Interface Modules", NRL Report 4385, November 1980.

[18] Parnas D.L., Wang, Y., "Simulating the Behaviour of Software Modules by Trace Rewriting Systems", *IEEE Transactions of Software Engineering*, Vol. 19, No. 10, pp. 750 - 759, 1994.

[19] Roubine, O., Robinson, L., "Special Reference Manual" Technical Report CSL-45, Computer Science Laboratory, Stanford Research Institute, August 1976.

[20] Robinson, L., Levitt K.N., Neumann P.G., Saxena, A.R., "A Formal Methodology for the Design of Operating System Software", in Yeh, R. (ed.), Current Trends in Programming Methodology I, Prentice-Hall, 1977, pp. 61-110.

[21] Robinson, L., Levitt K.N., "Proof Techniques for Hierarchically Structured Programs", *Communications of the ACM*, Vol. 20, No. 4, pp. 271-283, 1977.

[22] Robinson, L., Levitt K.N., Silverberg, B.A.,"The (HDM) Handbook", Computer Science Laboratory, SRI International, June 1979.